Installations and Very Large Systems

Eveliina Vuolli

University of Tampere
Department of Computer Science
Pro gradu thesis
December 1998

University of Tampere
Department of Computer Science
Eveliina Vuolli: Installations and Very Large Systems
Pro gradu thesis, 55 pages

December 1998

The importance of designing installations becomes all the time more important because of the software evolution and increasing amount of maintenance work done. Nowadays, software products and entire software systems are not installed only once but also upgraded several times during their lifetime.

First, this thesis introduces the characteristics of large software systems. Also special problems with installations of large systems are discussed. Then, the general principles in designing installations are introduced. The principles are represented in the form of requirements which are categorised in the following way: requirements for installation process, applications to be installed, installation procedures and installation tools. Also a system for dynamic installation of distributed software components is introduced. Together with the example environment and related problems the proposed system illustrates some of the requirements presented earlier.

It can be concluded that the basis for a successful installation is the application to be installed although the requirements for the installation process, procedures and tools should neither be forgotten. However, the installation can rarely fix the faults done during the software design and implementation.

Keywords: installation, upgrade, commissioning, large systems, software evolution, maintenance.

Acknowledgements

I wish to thank Nokia Telecommunications for the opportunity to carry out my pro gradu thesis in addition to my regular job. I have had a chance to enjoy a pleasant working atmosphere, and I wish to express my gratitude for everyone in my section, NMS Core System Solutions. Particularly my section manager and thesis instructor Juha Havulinna, as well as all the other people I have consulted deserve to be acknowledged for their helpful comments and ideas for my thesis. I also wish to thank the personnel of my group, NMS Installation and Upgrade (GRIP), for their patience and understanding during the project.

My special thanks go to my parents who have always supported me in my studies and my husband, Antti, for his encouragement and support during difficult times.

Finally, I would like to thank Ari Jaaksi for helping me to get started with my thesis, and professor Jyrki Nummenmaa at the University of Tampere who acted as my supervisor and gave me valuable advice and comments throughout the rest of the project.

Tampere, December 21st, 1998

Eveliina Vuolli (formerly known as Keskipoikela)

Table of contents

1. Introduction

Software evolution and maintenance are an increasing part of software engineering. Installations, especially upgrades, result from these two activities because recreated software products must be delivered again and again to the end users. The software evolution seems to be characteristic especially for large systems whose installations have even more problems than installations of smaller ones.

During the years, software-engineering theorists have listed various characteristics that express the quality of a software product. For example, according to Ghezzi et al. [1991] among the most important qualities of software process and products are maintainability, correctness, reliability, user friendliness, reusability, portability, understandability, and visibility. However, for installations or designing installations such guidelines are not available, mostly because in general, installations are not very often discussed in books in the field of software engineering. One of my goals in this thesis is to generate a similar list of qualities for installations; to list such characteristics, which would indicate the level of quality of an installation.

The research problems in this thesis are

1. What are the general principles in designing installations?
2. What are the problems with installations of large systems?
3. How can these problems be solved?

1.1 History of software evolution

There has not always been a need for mass installations of software products. The purpose of this section is to show how software systems have developed to their current state.

Table 1 illustrates the evolution of software within the context of computer-based system application areas. During the early years of the computer system development, hardware underwent continual change while software was viewed by many as an afterthought. Only a few systematic methods existed for computer programming and software development was virtually unmanaged. During this period, a patch orientation was used for most systems. Software was custom-designed for each application and had a relatively limited distribution. [Pressman, 1994] Software development was mainly a single-person task. The problem to be solved was well understood, and there was no distinction between the programmer and the end user of the application. The model used in these early days may be called the code-and-fix model. [Ghezzi *et al.*, 1991]

Table 1. Evolution of software (adapted from Pressman [1994])

The early years	The second era	The third era	The fourth era

• batch orientation • limited distribution • custom software	• multi-user • real-time • database • product software	• distributed systems • embedded "intelligence" • low-cost hardware • consumer impact	• powerful desk-top systems • object-oriented technologies • expert systems • artificial neural networks • parallel computing
1950 → mid-1960s	mid-1960 → late 1970s	mid-1970s →	1990s →

The second era of computer system evolution spanned the decade from the mid-1960s to the late 1970s. Multiprogramming, multi-user systems, real-time systems and first generation of database management systems were first introduced during this period of time. The second era was also characterised by the use of product software and the advent of "software houses". Software was developed for widespread distribution in a multidisciplinary market. As the number of computer-based systems grew, libraries of computer software began to expand and for the first time there was a need for software maintenance. Worse yet, the personalised nature of many programs made them virtually unmaintainable. [Pressman, 1994] This failure of the code-and-fix process model led to the recognition of the so-called software crisis and, in turn, to the birth of software engineering as a discipline. [Ghezzi *et al.*, 1991]

The third era of computer system evolution began in the mid-1970s and continues today. The distributed systems greatly increased the complexity of computer-based systems. Global and local area networks, high-bandwidth digital communications, and increasing demands for instantaneous data access put heavy demands on software developers. The third era has also been characterised by the advent and widespread use of microprocessors, personal computers, and powerful desktop workstations. The personal computer has been the catalyst for the growth of many software companies. While the software companies of the second era sold hundreds or thousands of copies of their programs, the software companies of the third era sell tens and even hundreds of thousands of copies. [Pressman, 1994]

Nowadays software is a product that must be marketed, sold, and installed on different machines at different sites. Because a sharp separation has arisen between software developers and end users, users must be trained and they must be assisted when something unexpected happens. Thus, economic, organisational and psychological issues have become important. In addition, demand has increased for much higher levels of quality in applications. Another sharp difference from the previous age is that software development has become a group activity. Group work requires carefully thought-out organisational structures and standard practices, in order to make it possible to predict and control developments [Ghezzi *et al.*, 1991]

The fourth era of computer software evolution is just beginning. Object-oriented technologies have taken their place in the field of software engineering, expert systems and artificial intelligence software have finally moved from the laboratory into practical application for wide-ranging problems in the real world. According to Pressman as we move into the fourth era, one of the problems intensifying in the computer software is that our ability to maintain existing programs is threatened by poor design and inadequate resources. [Pressman, 1994]

It can be seen very clearly that the importance of designing installations has become more and more important all the time. In the beginning of software evolution, when the software product was maybe installed only once on a single machine, the way it was done was not so important. Nowadays, software products and entire software systems are not only installed on large systems containing a network of computers but also upgraded several times during their lifetime. Thus, it is very natural that the way installations are implemented should be improved and the operational interruption should be minimised. When the amount of installations grows, it is also very reasonable to reduce the time spent in them because of the lack of resources, for example.

1.2 Terminology

The installation of a software product is not merely copying the software from CD-ROM to the machine's hard disk. Additionally, the environment where the application is installed usually needs some kind of adaptation. Jacsó [1992] defines the *installation process* in the following way: "the installation process accommodates an application in the prevailing environment". *Customisation* is another term which is closely related to installation. Jacsó's definition for customisation is "the process when some features of the application are tailor-made by the system administrator, such as the maximum duration of a search session".

In this document, different terms are used for different kinds of installations. The term installation includes both commissioning and upgrade [Setälä, 1998]. The term *commissioning* is used to refer to a new installation [Setälä, 1998]. In other words, in commissioning the software product is installed on a machine for the very first time. An *upgrade* is an installation where the older version of the software is replaced with the newer, presumably better, one [CCI Dictionary, 1998] [Setälä, 1998].

In this thesis, also *uninstallation* is included under the term installation. In the uninstallation the installed software or hardware is removed from the computer. Uninstallation includes both removing all the files that were installed and restoring all the modifications made to the system. [CCI Dictionary, 1998]

2. Software evolution and maintenance

2.1 Software evolution

On the contrary to the Section 1.1, this section tries to answer the questions, why software as a product evolves and how the evolution proceeds. Also Lehman's five laws of software evolution are introduced.

Successful software products are quite long lived [Ghezzi *et al.*, 1991]. According to Oskarsson [1982], an ideal system is a system with a long lifetime and availability, successively being adapted to new needs of users, new hardware, environmental changes and new applications. The first release of a software product is the beginning of a long lifetime and each successive release is the next step in the evolution of the system [Ghezzi *et al.*, 1991]. In other words, change is inevitable when computer-based systems are built [Pressman, 1994].

The need for future changes generates a strong economic pressure for flexibility in large software systems. Oskarsson gives two examples of reasons why the need for modifiability in software systems is well established [Oskarsson, 1982, p. 4]:

1. "Large software systems will have to be changed because their environments and their applications will change during the systems' life-times, even if the software systems were intended for only one application and one environment each from the beginning.
2. The pressure for cost-effective software leads to design of software systems intended for large classes of applications and for a variety of environments. However, it is neither possible to foresee nor reasonable to implement all the relevant applications and environments when basic design is performed. Usually, the aim is to make a modifiable system which can be adapted for new applications and environments when needed."

How does the evolution then proceed? Clearly, it is not a natural process governed by unchangeable laws of nature. However, there are regularities, trends, and patterns that appear and dominate evolution of large systems. These common features and patterns of behaviour reflect common characteristics from which laws can be deduced according to Lehman. The laws, derived from experimental observations of a number of systems, have also very practical application. They provide a basis for life cycle management tools of large software systems, as well as insight and understanding for improvement of the programming process. [Lehman, 1978b]

The five laws of program evolution are Continuing change, Increasing complexity, The fundamental law, Conservation of organisation stability and Conservation of familiarity. [Lehman, 1978b, p. 381]

1. *Continuing change.* All programs are models of some part or aspect of the real-world environment that undergoes continuing change. Thus, the program itself must also change or it becomes less and less useful in that environment. [Lehman, 1978b]
2. *Increasing complexity*. The complexity of an evolving program increases all the time because of the deteriorating structure unless work is done to maintain or reduce it [Lehman, 1978b]. It should be noted that the program complexity is relative to a level of perception [Lehman, 1978].
3. *The fundamental law of program evolution*. Program evolution is a self-regulating process and measurement of system attributes such as size, time between releases, number of reported errors, etc. reveals statistically significant trends and invariances [Sommerville, 1982]. This law is also called *the Law of Statistically Smooth Growth* [Lehman, 1978b].
4. *Conservation of organisation stability*. During the lifetime of a program, the rate of its development is approximately constant and independent of the resources devoted to system development [Sommerville, 1982]. This law is also called *the Invariant Work Rate* [Lehman, 1978b].
5. *Conservation of familiarity*. The content of each successful release including changes, additions and deletions is approximately constant during the lifetime of an evolving program. This law has also been called *The Law of Incremental Growth Limits.* [Lehman, 1978b]

Sommerville [1982] points out that the Lehman's laws are not universally accepted but that they do appear to have some validity in many cases. In fact, in the experimental study of Lawrence [1982] supporting data was found for the first two laws while there was no supporting data for the last three laws. Also Lehman [1978b, p. 393] says that

> "… they (laws) stand in their own right until accumulating evidence and developing insight and understanding demand their change – or until we can so change the system structure, process methodology and characteristics, and programmer and user practice and habits, that the laws as formulated no longer apply."

Because Lawrence strongly thinks that there is little reason to intuitively expect Lehman's last three laws to be true he suggests that the environment in which a system exists and the way it responds to its environment is complex. In addition, it is likely to be system and organisation specific. Contradictory to Lehman's ideas, Lawrence states that as systems are used they evolve by physically growing in size but that the dynamics of the growth process look to be discontinuous and quite irregular. Thus, he searches for principles governing the growth process beyond the software product itself. The factors influencing systems evolution are exhibited in Figure 1. [Lawrence, 1982]

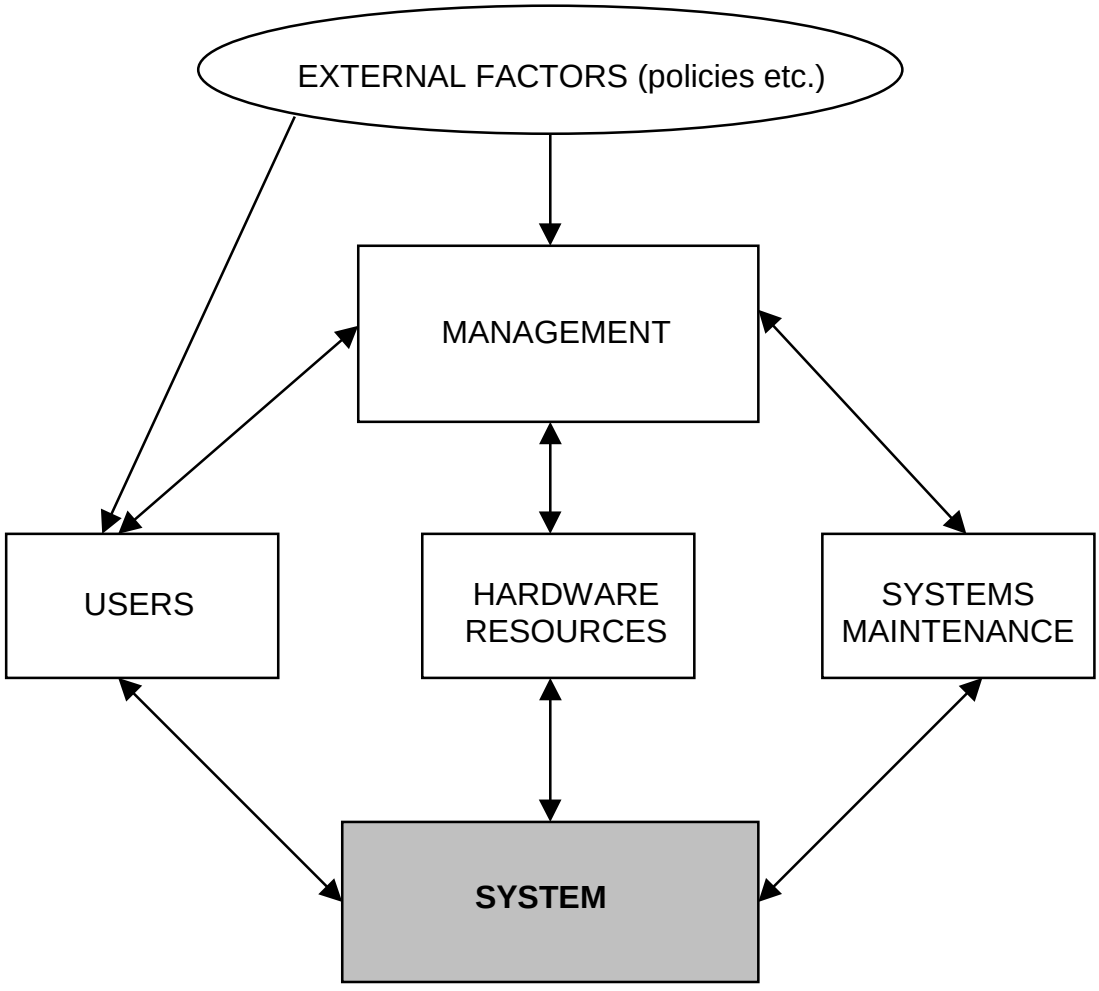

Figure 1. Model of systems evolution (adapted from [Lawrence, 1982])

2.2 Software maintenance

Software evolution and maintenance are closely related concepts. It could be said that maintenance work is a consequence of software evolution. Schneidewind [1987, p. 303] offers a more detailed definition for maintenance:

> "Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment."

The area of software maintenance has been described as an "iceberg" mainly because an enormous mass of potential problems and cost lies under the surface [Pressman, 1994] [Swanson, 1976]. Additionally, Pressman [1994] points out that software maintenance has until very recently been the neglected phase in the software engineering process. The literature on maintenance contains very few entries when compared to development activities and relatively little technical approaches or methods have been proposed [Pressman, 1994].

According to Pressman [1994] the maintenance of existing software can account for over 70 percent of all effort expended by a software organisation. Also Lientz and Swanson's study [1980] showed that the time consumed in the maintenance work was about the half from the total work amount. However, the larger organisations tended to spend more time on maintenance than smaller ones [Lientz and Swanson, 1980].

It is worth noticing that the amount of time spent by an organisation on software maintenance places a constraint on the effort that may be put into new system development. Further, where programming resources are cut back due to economic pressures, new development is likely to suffer all the more, since priority must be given to keeping current systems up and running. [Swanson, 1976]

In the following Section 2.2.1 different maintenance tasks are listed. Maintenance costs are dealt with in Section 2.2.2. Section 2.2.3 discusses problems occurring in the maintenance.

2.2.1 Maintenance activities

Usually the modifications performed on software are repair, i.e. redesign, enhancement and tuning [Belady, 1980]. However, software maintenance is far more than "fixing mistakes" [Pressman, 1994]. Swanson [1976] describes three maintenance activities that are undertaken after software product is released for use: corrective, adaptive and perfective maintenance.

Maintenance performed in response to processing, performance and implementation failures may be termed *corrective maintenance* [Swanson, 1976]. Corrective maintenance is an activity which would not be performed at all, if the software testing could cover all latent errors in a large software system [Pressman,

1994] [Swanson, 1976]. Thus, its costs must be compared with the opportunity costs of implementing more "failure-free" software [Swanson, 1976].

Maintenance performed in response to changes in data and processing environments may be termed *adaptive maintenance* [Swanson, 1976]. The useful life of application software is much longer than the life of the system environment for which it was originally developed [Oskarsson, 1982] [Pressman, 1994]. The timely anticipation of environmental change is necessary to ensure effective performance of this type of maintenance [Swanson, 1976]. The amount of adaptive maintenance which must be performed on software is often a reflection of program portability, i.e. the transferability of the program to new data and processing environments [Swanson, 1976].

In contrast to corrective and adaptive maintenance, which serve merely to keep a program up and running, *perfective maintenance* is directed toward keeping a program up and running at less expense, or so as to better serve the needs of its users [Swanson, 1976]. In other words, this activity does not originate from an unsuccessful software package. Instead, as the software is used, recommendations for new capabilities, modifications to existing functions, and general enhancements are received from users [Pressman, 1994]. This activity accounts for the majority of all effort expended on software maintenance [Pressman, 1994].

In addition to these, Pressman defines a fourth activity, which is *preventive maintenance*. The fourth maintenance activity occurs when software is changed to improve future maintainability or reliability, or to provide a better basis for future enhancements. This term is commonly used in the maintenance of hardware and other physical systems. It should be noted, however, that analogies between software and hardware maintenance can be misleading. [Pressman, 1994]

According to a study by Lientz and Swanson [1980], about 55 % of maintenance is perfective, 25 % adaptive and 20 % corrective. The time spent on different kinds of maintenance tasks according to the study is represented in Figure 2. Unfortunately they do not recognise preventive maintenance. Also Pressman [1994] does not provide estimate as to how large a portion the preventive maintenance takes from the total maintenance effort.

Some software professionals are troubled by the inclusion of the second and third activities as a part of a definition of maintenance [Pressman, 1994]. For example Lehman [1982] states that the use of term maintenance should be abandoned and replaced with evolution. It is true that the tasks that occur as a part of adaptive and perfective maintenance are the same tasks that are applied during the development phase of the software engineering process [Pressman, 1994]. However, such tasks, when they are applied to an existing program, have traditionally been called maintenance [Ghezzi *et al.*, 1991][Pressman, 1994].

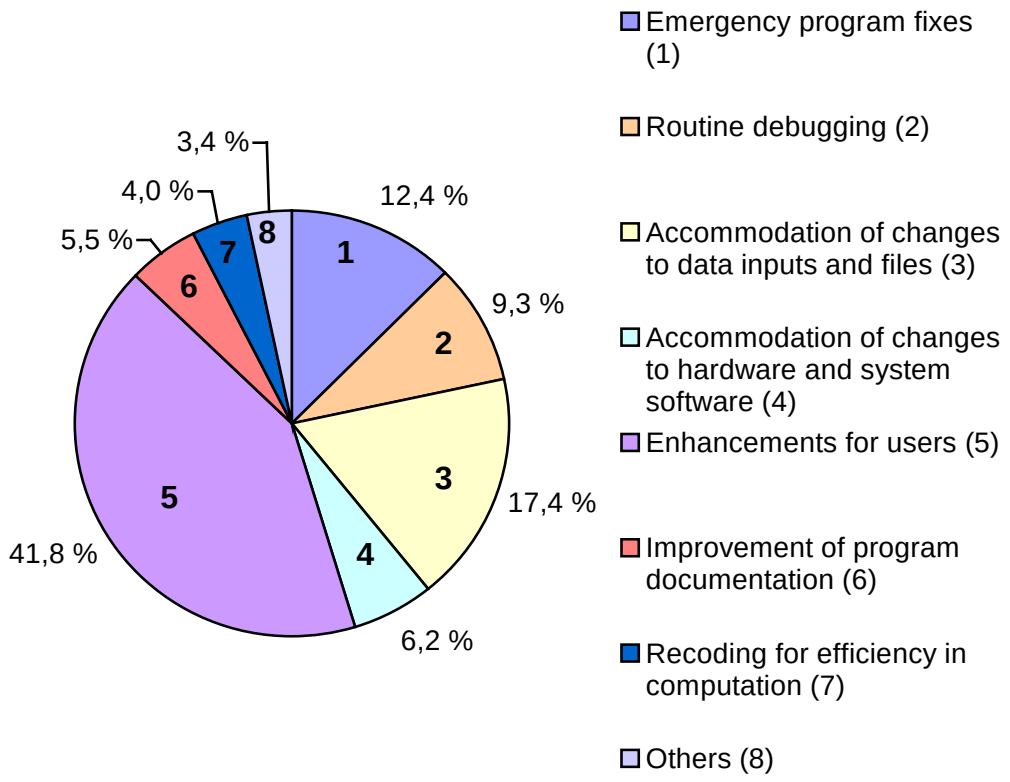

Figure 2. Percentage of hours expended on different maintenance tasks

Oskarsson [1982] separates modification and normal maintenance activities such as error correction and optimisation. He refers to this continuous development and adaptation by the term software evolution discussed also in the previous Section 2.1 [Belady and Lehman, 1979] [Oskarsson, 1982]. Examples of modifications are extension of functional capabilities, removal of functional capabilities, adaptation to new processor hardware and adaptation to new environments [Oskarsson, 1982].

Extension of capabilities includes both the extension of an existing capability and the introduction of completely new capabilities. [Oskarsson, 1982]

Deletion of unwanted capabilities is not always worthwhile. In many cases the costs in memory size and execution time are minor compared to the cost of removing the extra program code, especially when dealing with capabilities whose realisations are woven in throughout the system. On the other hand, the "leftover" capabilities tend to increase the complexity of systems, thus making debugging and further evolution more difficult. In general it is more difficult to remove the implementation of a capability than to introduce a new one. When introducing a new capability, the designer can choose to connect it to the system in the simplest possible way. When removing the realisation of an unwanted capability, there is no such choice. [Oskarsson, 1982]

Adaptations to environments are required in two ways. First, new areas of application will include new environments to which adaptation must take place. Examples are personnel with new needs, and new types of external hardware equipment. Second, technological and social development will affect existing systems. New hardware devices are connected to existing software systems, new laws and new demands from labour unions will change the system's interaction with its environment. An example is new security requirements. It is not always clear if a new requirement concerns adaptation to the environment or extension of capability. An adaptation may very well be realised through a new capability. [Oskarsson, 1982]

Adaptation to new processor hardware is most likely needed if a software system is to live and be delivered to customers for twenty years. Also in a shorter perspective, many types of systems are moved to different computers. This concerns the problem of software portability. The difference to previous adaptation is not always distinct. For example, to an operating system a tape driver may be a part of the environment, whereas for an application program it may be a part of the processor hardware. [Oskarsson, 1982]

Cleaning-up means keeping the handling qualities of the system at an acceptable level which is a large part of the evolution effort. These activities are usually not directly visible for the users, and are often performed for the vendor's own sake. [Oskarsson, 1982]

In my opinion, the difference between the terms "software evolution" and "maintenance" is not so crucial. First, there will never be an ideal situation where all the errors in the software would be detected before it is delivered to the customers. Thus, error correction will be done among other maintenance activities in the future. Secondly, it is not so easy to make a difference between the maintenance activities although they were presented as well-defined concepts. Sometimes, when a designer corrects an error he might at the same time implement an enhancement to another part of the same module. Is this perfective or corrective maintenance? Equally in addition to error corrections a designer could implement a totally new feature to the module. Again, it is very difficult to say if the work done is corrective maintenance or extension of capabilities.

2.2.2 Maintenance costs

The cost of software maintenance has increased steadily during the past 20 years which is a natural progress considering the history of software evolution (see Section 1.1). During the 1970s, maintenance accounted for between 35 and 40 percent of the software budget and information system organisation. This number jumped to approximately 60 percent during the 1980s. [Pressman, 1994]

Pressman points out that even though the financial cost of maintenance is the most obvious concern there are other less tangible costs that should be thought of. The intangible costs of software maintenance include

- development opportunity that is postponed or lost because available resources must be channelled to maintenance tasks,
- customer dissatisfaction when seemingly legitimate requests for repair or modification cannot be addressed in a timely manner,
- reduction in overall software quality as a result of changes that introduce latent errors in the maintained software,
- upheaval caused during development efforts when staff must be "pulled" to work on a maintenance task and
- a dramatic decrease in productivity that is encountered when the maintenance of old programs is initiated. [Pressman, 1994]

Estimating maintenance costs for any particular program is very difficult. The difficulties arise because these costs are related to a number of relatively unpredictable factors. According to Sommerville these include the application being supported, staff stability, the lifetime of the program, the dependence of the program on its external environment and hardware stability. Maintenance costs are also governed by less unpredictable, technical factors. [Sommerville, 1982]

2.2.3 Maintenance problems

Why, in the first place, are there problems with maintenance? Most authors accuse the lack of control and discipline of maintenance problems. In the maintenance work, same kinds of methods should be used as in the actual software design [Belady, 1980] [Ghezzi *et al.*, 1991] [Pressman, 1994]. Another important issue is that the specifications, i.e. documentation, are not updated to reflect the change [Belady, 1980] [Ghezzi *et al.*, 1991]. Belady [1980] claims that this is due schedule and cost pressures directed towards maintenance groups. Within very tight schedules the design documents are left unchanged, reflecting the state in which the software was, and not the state in which the software is [Belady, 1980]. Unfortunately, this neglect makes future changes, i.e. maintenance work, more and more difficult to apply [Belady, 1980] [Ghezzi *et al.*, 1991] and forces the crew to work with the only trustworthy document, the low-level code itself [Belady, 1980].

The maintenance work done without proper documentation is often referred as unstructured maintenance. If a complete software documentation exists, the maintenance is structured. Although the existence of up-to-date documentation does not itself guarantee problem-free maintenance, the amount of wasted effort is reduced and the overall quality of a change or correction is enhanced. [Pressman, 1994]

According to Schneidewind, the main problem in doing maintenance is that we cannot maintain a system which was not designed for maintenance. He lists also other classic problems that can be associated with software maintenance. The problems are concluded in the following way by Pressman [1994, p. 680]:

- "It is often difficult or impossible to trace the evolution of the software through many versions or releases. Changes are not adequately documented.
- It is often difficult or impossible to trace the process through which software was created.
- It is often exceptionally difficult to understand "someone else's" program. Above all, if only undocumented code exists, severe problems should be expected.
- "Someone else" is often not around to explain. Mobility among software personnel is high. We cannot rely upon a personal explanation of the software by the developer when maintenance is required.
- Documentation does not exist or is awful. The recognition that software must be documented is a first step, but documentation must be understandable and consistent with source code to be of any value.
- Most software is not designed for change. Unless a design method accommodates change through concepts such as functional independence or object classes, modifications to software are difficult and error-prone.
- Maintenance has not been viewed as very glamorous work. Much of this perception comes from the high frustration level associated with maintenance work."

In general, maintenance work is viewed as a tedious job. Maybe that is why Belady [1980] claims that the people selected for maintenance work are less skilled than the other software designers. Naturally, this can cause even more problems.

Studies of large software systems also show that evolvability decreases with each release of software product as Lehman's second law states. Each release complicates the structure of the software, so that future modifications become more difficult [Belady, 1980] [Belady and Lehman, 1979] [Ghezzi *et al*., 1991] [Lehman, 1978b]. The reason for this is that the original structure which was well matched to the original requirements is gradually deteriorating as more recent requirements induce changes which do not fit the old structure [Belady, 1980]. Since one must live with an ever changing environment, one must explore methods which at least reduce this deterioration of structure during unavoidable program evolution [Belady, 1980] [Belady and Lehman, 1979].

The alternative is to reach such a level of complexity that further evolutionary progress can only be made through re-creation [Belady and Lehman, 1979]. Most software systems start out being evolvable, but after years of evolution they reach a state where any major modification runs the risk of "breaking" existing features [Ghezzi *et al*., 1991]. Belady and Lehman [1979] note that the technological aim must be to achieve the most economic balance between continuous or periodic restructuring and periodic recreation [Belady and Lehman, 1979].

2.3 Software system elements and modification tasks

What kinds of tasks originate from maintenance activities? Before we can get a closer look at this question, we have to understand which elements are included in a software product or system. Pressman [1994, pp.132-133] defines the elements of a computer-based system (illustrated in Figure 3) in the following way:

- "Software: Computer programs, data structures, and related documentation that serve to effect the logical method, procedure or control that is required
- Hardware: Electronic devices (e.g., CPU, memory) that provide computing capability, and electromechanical devices (e.g., sensors, motors, pumps) that provide external world function
- People: Users and operators of hardware and software
- Database: A large, organised collection of information that is accessed via software and is an integral part of system function
- Documentation: Manuals, forms, and other descriptive information that portrays the use and/or operation of the system
- Procedures [1]: The steps that define the specific use of each system element or the procedural context in which the system resides"

It should be noted that the software installation is only a part of installing a whole software system. It might be better to talk about system upgrades instead of software upgrades because the system upgrade can also include hardware and database changes needed in some stage of the system evolution.

When comparing the system elements to different maintenance activities described in Section 2.2.1, the following conclusions can be made regarding system upgrades as a whole instead of software upgrades. Corrective maintenance includes only software modifications because it is due to software errors. In this case, if no other types of maintenance activities are needed, only software needs to be upgraded.

[1] Note the different meaning of the term "procedure" in this context and in other parts of the thesis.

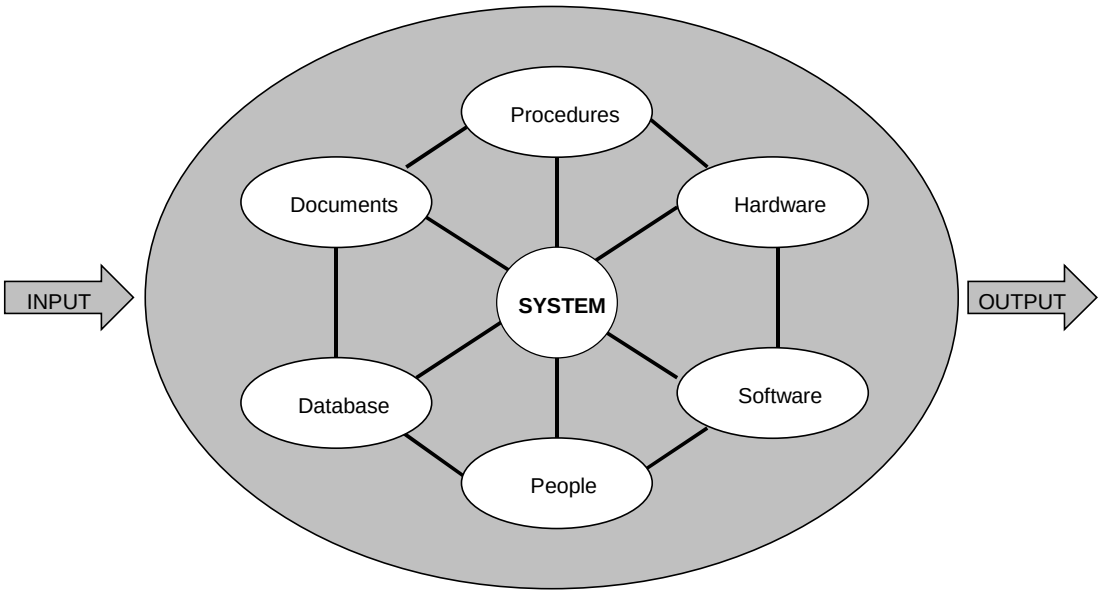

Figure 3. System elements [Pressman, 1994]

Instead, adaptive maintenance causes more extensive upgrade because software changes are made just because of changes e.g. in the hardware. This means that at the same time both software and hardware must be upgraded. Perfective and preventive maintenance can concern only software but on the other hand they can also include upgrading hardware and database at the same time as a preparation for increasing future needs. I think that in practice all the elements of the system are usually affected in the upgrades, especially in the upgrades of large software systems.

Oskarsson [1982] discusses the efforts required for modifications especially in the software element. According to him the main tasks are Understanding where to change, Analysing ripple effects, Performing software modifications, and Testing system behaviour.

Understanding where to change. The person responsible for implementing a change for a certain new requirement must find out where, and in what way, the concerned capabilities are implemented. It should be noted that this person is presumably not the one who designed the software originally. [Oskarsson, 1982]

Analysing the ripple effects. Because of dependencies between different parts of the software, the devised change may introduce inconsistencies in dependent software. The corrections to these inconsistencies may create new inconsistencies, and corrections can thus spread as ripple effects through the system. Dependencies between software units must be understood so that such ripple effects can be tracked down. [Oskarsson, 1982]

Performing the software modifications. According to Oskarsson [1982] the main effort required when changing software is not the manual task of editing source-code. It is more difficult to manage the updating process (especially if several persons are involved), to understand what one is doing, and to perform and document it in a

correct way. A complicated task can be very difficult in this way, even if it is completely well defined. [Oskarsson, 1982]

Testing system behaviour. It must be tested to ensure that the required changes in system behaviour have been effected and that all other aspects of the system are left unchanged. In an evolving system, it should be easy to test capabilities that were not to be changed, since old test data can be used. [Oskarsson, 1982]

It should be noted that all these tasks become significantly easier if the following simple maintenance practices are used: easiest change is made first and only one module is changed at a time. [Schneidewind, 1987]

3. Large systems

The development of large systems differs considerably from that of small ones, although the typical examples in software engineering books consider relatively small applications [Aalto, 1995]. This also affects the installation; the larger the software product increases, the more complicated the installation becomes. What is then "a large system"? How does it differ from smaller ones, especially with regard to installations? Definitions for a large system are presented in Section 3.1. The special characteristics of large systems are described in Section 3.2, and Section 3.3 discusses their influence on installations according to tasks presented in Section 2.3.

3.1 Project size categories

Fairley [1985] categorises the project sizes in the following way: trivial, small, medium-size, large, very large and extremely large projects (see Table 2).

Table 2. Size categories for software products, adapted from Fairley [1985]

Category	Number of programmers	Duration of software development	Product size	Interactions to other programs
Trivial	1	1-4 weeks	500 source lines	none
Small	1	1-6 months	1K-2K	none
Medium	2-5	1-2 years	5K-50K	none/few
Large	5-20	2-3 years	50K-100K	significant
Very large	100-1000	4-5 years	1M	complex
Extremely large	2000-5000	5-10 years	1M-10M	complex

According to Fairley, *a trivial software project* involves one programmer working for a few days or a few weeks and results in a program of less than 500 statements, packaged in 10 to 20 subroutines. Such programs are often personal software and are usually discarded after a few months. There is little need for formal methods in designing and implementing trivial programs. However, one of the dangers of this kind of software development is that a program intended for personal use only becomes a software product, but without the benefit of the planning and support required for a product. [Fairley, 1985]

A small project employs one programmer for 1 to 6 months and results in a product containing 1000 to 2000 lines of source code packaged in perhaps 25 to 50 routines. Small programs usually have no interactions with other programs. Fairley's list of examples of such programs include scientific applications written by engineers

to solve numerical problems, small commercial applications written by data processing personnel to perform straightforward data manipulation and report generation, and student projects written in compiler and operating system courses. [Fairley, 1985]

A project of medium size requires two to five programmers working for 1 to 2 years and results in 10 000 to 50 000 lines of source code packaged in 250 to 1000 routines. Products of medium size have few, if any interactions with other programs. Medium-size programs include assemblers, compilers and small management information systems. According to Fairley the vast majority of software projects, and the resulting programs, are of small or medium size. [Fairley, 1985]

A large project requires 5 to 20 programmers for a period of 2 to 3 years and results in a system of 50 000 to 100 000 source statements, packaged in several subsystems. A large program often has significant interactions with other programs and software systems. Examples of large programs include large compilers, database packages and real-time control systems. [Fairley, 1985].

A very large project requires 100 to 1000 programmers for a period of 4 to 5 years and results in a software system of 1 million source instructions. A very large system generally consists of several major subsystems, each of which forms a large system. The subsystems typically have complex interactions with one another and with other separately developed systems. Very large systems often involve real-time processing, telecommunications, and multitasking. Examples of such systems include large operating systems, large database systems, and military command and control systems. [Fairley, 1985]

An extremely large project employs 2000 to 5000 programmers for periods of up to 10 years and results in 1 million to 10 million lines of source code. Extremely large systems consist of several very large subsystems and often involve real-time processing, telecommunications, multitasking, and distributed processing like very large projects. Extremely large systems also often have very high reliability requirements and involve life-and-death processes. Examples of extremely large systems include air traffic control, ballistic missile defence, and military command and control systems. Fairley points out that very few extremely large systems have been built so far but that increasing pressures of modern society and new advances in software technology will result in increasing numbers of extremely large projects in the future. [Fairley, 1985]

Belady and Lehman [1979] approach the concept of large system quite differently. According to them the largeness of a software system is not so much a question of number of modules in the system as the number of people involved in the specification, design, testing, maintenance and operation of the system. Also Oskarsson [1982] views this as being the most important criterion for largeness where modifiability is concerned. Belady and Lehman [1979] state that the root cause of the emergence of the largeness characteristics they identify is related to the concept

variety. The variety can occur in the program's operational environment or in the human interests and activities the program reflects [Belady and Lehman, 1979].

In his other article, Lehman states that a program is large, if it requires, or is given, an organisation of at least two levels of management for its development or maintenance. When such a project organisation exists, the phenomena associated with largeness will appear. [Lehman, 1978]

3.2 Characteristics of large systems

According to several authors, the main difference between small and large systems is that large systems are complex and require so much effort that they cannot be created by one individual and thus organised teams are needed [Aalto, 1995] [Belady and Lehman, 1979] [Fairley, 1985]. Division of labour is very characteristic to organisations developing large systems because it is the most cost effective manufacturing process [Belady and Lehman, 1979]. Teams can specialise in architectural issues, design, programming, database management, documentation, or testing, for example [Belady and Lehman, 1979].

Good communication is essential both between and within these teams [Aalto, 1995] [Belady and Lehman, 1979] [Curtis *et al*., 1988] but unfortunately even the formal communication structure can be a hindrance for communication [Curtis *et al*., 1988]. Communication difficulties can also be encountered due to geographic separation, cultural differences, and environmental factors [Curtis *et al*., 1988]. In addition, there is a great likelihood that there will be some turnover of project personnel during the development cycle [Fairley, 1985]. This will require training of new personnel or distribution of the responsibilities of a departed team member among the remaining members [Fairley, 1985].

Large systems are typically developed incrementally and new features are developed by reusing existing software. This is mainly due to normal software evolution discussed in Section 2.1. Both customers and developers may initiate changes to the product requirements as the project evolves. [Aalto, 1995] [Belady and Lehman, 1979] [Fairley, 1985]

Large systems are often mission critical systems, such as air traffic control systems and telecommunications infrastructures. They are extremely important for the customer's business, and the quality of the delivered systems needs to be particularly high to ensure maximum reliability. In addition, the customers of companies developing large systems are typically other companies, not consumers. Therefore, the large systems need to fit into the business activity of the user organisation. [Aalto, 1995]

The last two characteristics are also related to the problem of thin spread of application domain knowledge. In the study of Curtis et al. this was found to be particularly characteristic of projects where software was embedded in a larger system

(e.g., avionics or telephony). These systems contrast with applications currently taught in computer science departments that do not require integration of various knowledge domains. As a result, software development requires a substantial time commitment for learning the application domain although individual staff members understood well different components of the application. [Curtis *et al.*, 1988]

3.3 Modifying and installing a large system

If one thinks about the main tasks needed to implement a modification to a software product (see Section 2.3), it can be seen very easily that the larger the software system grows, the more difficult modifications become. Understanding where to change, analysing the ripple effects and performing the software modifications need much more effort when the system, which should be understood and to be analysed, becomes larger. Because one individual cannot have control of the whole system, co-operation is needed in all the listed tasks. One cannot stress enough the importance of good communication and documentation.

In addition, installations of large systems are more complicated than installations of small ones because they have complex interactions to other programs, they are usually mission critical systems and they are developed by a group of persons instead of a single designer.

The interactions with other programs are quite probably affected when a software product is installed on a machine. If the installation is a commissioning, it might be necessary to install other programs at the same time because otherwise the software would not work. Often these products are so called 3rd party software made by another software company and they might be related to databases or communications protocols. In an upgrade, the interactions with other programs must be known because the connections between programs must work afterwards. The other software products might also need upgrading or the configuration between the programs should be modified during the upgrade.

Because large systems are usually mission critical for the customer, their installations are required to be more reliable and faster. This subject is discussed under the term "downtime" in Section 4.2.4.

When designing and implementing an installation, the larger the software system is the more persons need to be involved. Good communication and documentation is needed once again because one person cannot "just know" what things are needed to make the system work. The information for installation must be collected from the design documentation and software designers themselves.

4. Principles in designing installations

There are several reasons why we should concentrate on the quality of installations and how the installations are implemented. In general, little or no considerations are given to the issue of installing the software on the customer machine(s) in a classic software design [Simmons, 1997]. Jacsó [1992] even claims that installation is usually the most dreaded part of introducing new CD-ROM products and services and it is often the first disappointment for the end user. He also stresses that the installation is not a one-time process. Each time a new version of the software is released you may have to go through the installation process again [Jacsó, 1992].

According to Hoek et al. the advent of the Internet and the use of component-based technology together have led to a radically new software development process: increasingly, software is being developed as a "system of systems" by a federated group of organisations. This leads to new kind of problems in such tasks as system installations, updates, and removals. In other words, when dealing with "system of systems" the importance of the quality of the installation increases. [Hoek *et al.*, 1997]

According to Jacsó, even if you have all the hardware and software requirements you may experience problems during, and most typically after, installation. Regretfully, the installation of a new product not only may be unsuccessful in itself, but also may prevent other, previously problem-free applications from running. [Jacsó, 1992]

Simmons lists some benefits of the increased installability. The ease of installation makes test runs and product sampling easier. It reduces the need of support during the installation and also at a later date if in a problem situation there is need to verify correctness of the installation. Additionally, future sales become more likely because of the increased value of the product to the customer. [Simmons, 1997]

What could be done to prevent the problems during the installations and make the installations more pleasant? I have collected some requirements and principles that help in designing better installations. Even though in the field of software engineering there is not much literature about installations, I found some articles that deal with the problem of installation. The problem with these references was that they were written from very different aspects. Some of them were concerned only about applications delivered on CD-ROM [Beiser, 1994] [Jacsó, 1992] or designed for Windows platform [Schiele, 1995] [Logo Guidelines, 1998] [Logo Handbook, 1998] while some of them were interested in how to modify programs on-the-fly [Amador *et al.*, 1991] [Fabry, 1976] [Segal and Frieder, 1993].

The requirements and principles do not necessarily improve the quality of the installations but will make them more predictable [Simmons, 1997] or they might make the system more transparent to both its users and programmers and its execution

environment [Segal & Frieder, 1993]. It should also be noted that all the requirements do not apply to all the installations [Segal & Frieder, 1993] and there may be reasons to violate one or more of them [Simmons, 1997]. The relative importance of the requirements varies with the application domain and the desired performance and correctness guarantees [Segal & Frieder, 1993].

4.1 Categorization

In my opinion, the requirements for installations have different levels. The highest level requirements can be directed towards the installation process itself. For example, it can be required that the installation does not have any influence on other software products or that there is no downtime during the installation. The demands for the dreamed installation process can also lead to new lower level requirements which I have divided into three separate categories: application to be installed, procedures implementing the installation and used installation tool. As a result of this classification, the requirements for installations can be divided into four different

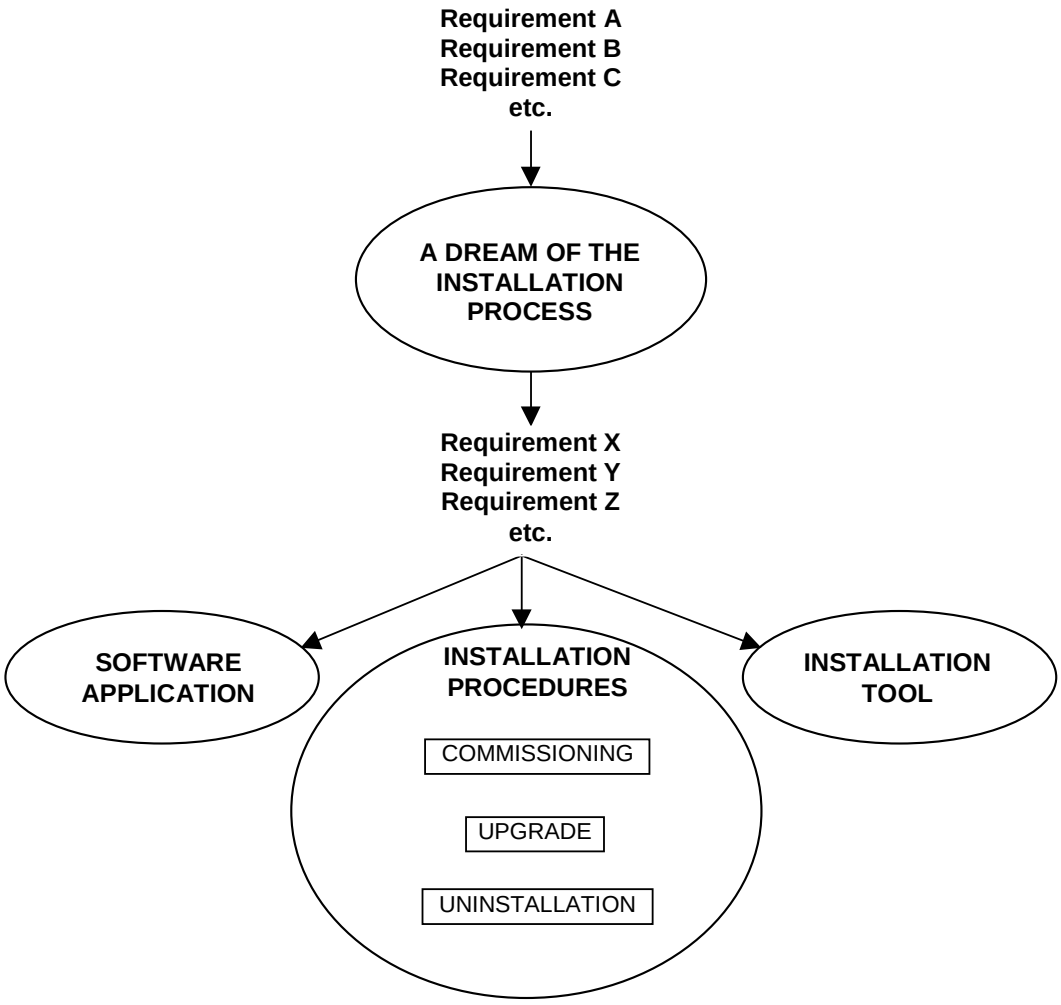

Figure 4. Categories for installation requirements

categories which are represented in Figure 4. Additionally, the installation procedures can implement either commissioning, upgrade or uninstallation, each of which can naturally have different kinds of requirements.

The requirements for the four categories are listed in the following Chapters. First the requirements for the installation process as such are described in Section 4.2. Section 4.3 describes how the applications can be made more installable. The requirements for commissioning and upgrade procedures are described in Section 4.4. The uninstallation procedures are discussed separately in Section 4.5. Section 4.6 includes requirements for the installation tool. Finally the common requirements for applications to be installed, installation procedures and tools are introduced in Section 4.7.

4.2 Requirements for the installation process

The requirements for the installation process itself are not very complex. In general, installations should be kept simple (see Section 4.2.1), they should offer alternate options for different situations (see Section 4.2.3) and their effect on the environment should be minimal aside from the installed product (see Sections 4.2.2 and 4.2.4). Also some organisational requirements are discussed in Section 4.2.5.

4.2.1 Simplicity

The installation process should be kept as simple as possible [Bianco *et al.*, 1994]. According to Bianco et al. [1994], it is impossible to overrate the importance of this guideline. The more complex the installation process is, the more difficult it is to maintain, debug and enhance, not to mention the fact that system administrators are less likely to trust the installation if they do not understand how it works [Bianco *et al.*, 1994]. To keep the installation simple, no special-purpose hardware should be required during the installation. Instead, the existing machinery should be exploited [Segal & Frieder, 1993].

There is also one very simple requirement, which should definitely be met in all installations. All the installed components must work after installation including shortcuts in the Windows environment, for example. [Logo Guidelines, 1998] [Logo Handbook, 1998]

The last requirement describes the current situation of the installations and how much installations have been thought of previously. If we did not want the application to work in the first place why would we even bother to make an uncompleted installation for it? The software vendors should not see installations as an unavoidable nuisance but as a service to their customers. Even financially, the installations are not just a burden. It is always possible to charge for a good service.

4.2.2 Minimal disruption

One of the most important requirements for installations is that the installation can be carried out with minimal disruption to both the environment and the software product [Simmons, 1997]. It should be noted that the product to be installed is not the only one on the workstation and other software products should not be affected during the installation [Jacsó, 1992], including the previous version of the product in question. In other words, it should be possible to upgrade the application without affecting the previous installation [Simmons, 1997]. It is also recommended that the installation process permits simultaneous installations [Simmons, 1997] [Lawson, 1994]. In my opinion, it is not so useful to have several versions of one software product installed because, most likely, only one of them is used. Therefore this requirement for saving and not affecting the older version is not very critical. However, during the installation there should always be a way back to the older version if the installation fails or the end user changes her mind about taking the new version into use.

As little disk space as possible should be occupied during the installation [Lawson, 1994]. In general, asocial behaviour and selfish resource allocation conventions should be avoided [Jacsó, 1992]. This means that the installation process should respect the existing technical and political policies of the environment, make minimal impact on the configuration of the machines on which it runs, and should not attempt to enforce change of style or environment on the end users or systems managers [Simmons, 1997]. If the application is a mission critical system, it can be even required that the program's performance should not be degraded during the upgrade [Segal & Frieder, 1993].

When discussing performance issues, there are always different viewpoints which should not be mixed together. The performance of all the applications during the installation is a different issue from the performance decrease or increase which is due to a new version of a specific software product. In general, I think the aim is always that the performance should not decrease when a new version of a software product is taken into the use. Another question is, whether the possible performance decrease is due to the new version itself or due to mistakes made during the upgrade.

After an upgrade, it might not be very easy to find out what is the real cause of the performance decrease. It might not even be clear if there is a decrease in the performance of a program. For example, the software product might not have worked well before the upgrade. If the customer does not notice it until afterwards, it is possible that he may blame the upgrade for everything. To avoid conflicts due to this kind of situation it might be a good idea to measure the performance before and after upgrade with some kind of tool. This would clarify the situation and the real cause for performance decrease, as well as for other kinds of problems, which would be easier to find.

Also system reboots should be avoided. If the system must be rebooted, installation should gracefully proceed where it left off [Logo Guidelines, 1998] [Logo Handbook, 1998]. The system reboot itself is not very dangerous but it takes time and during reboot the system is not available for the users.

A proper installation process does not leave any traces of itself other than the just installed application. All temporary files and other such things should be removed after installation [Jacsó, 1992]. Applied to the uninstallation, this requirement means that the uninstaller must also remove itself during the uninstallation [Logo Guidelines, 1998] [Logo Handbook, 1998].

4.2.3 Alternative options

This section proposes such requirements which demand the installations to adapt themselves to different kinds of situations. The alternative can concern e.g. the type of installation, the setup options, the level of automation and assistance the installation procedures and tools offer, the remoteness of an installation, and the type of media in which the software product and installation is delivered to end users.

An installation tool should not be designed to accomplish only commissionings [Schiele, 1995]. Instead, it should be prepared to update [Schiele, 1995] or remove an already installed application [Logo Guidelines, 1998] [Logo Handbook, 1998] [Schiele, 1995]. The easiest way to implement this requirement is that the installation tool itself should not change in different situations but only makes the decision as to which one of the separately provided procedures (commissioning, upgrade and uninstallation) is run.

The installation should also offer different kinds of setup options [Schiele, 1995]. This means that in addition to a typical installation, a predefined subset of the software (compact installation) can be installed or the end user can choose the parts to be installed (custom installation) [Schiele, 1995]. In the compact installation, usually the smallest possible set of the software product is installed. The custom installation can include only one or more subproducts that the end user wants to have and it is possible to install more products later.

Segal and Frieder [1993] strongly emphasise that the human intervention should be minimised during the installations. According to them, only the fully automated installation ensures that the updating components are applied in the correct order and at the right time [Segal & Frieder, 1993]. Also Lawson [1994] states that the installation process should be totally automated. However, other authors seem to think that the choice of silent, i.e. automated, or attended installation should be left to the person accomplishing the installation and so both alternatives should be provided [Logo Guidelines, 1998] [Logo Handbook, 1998] [Schiele, 1995]. In my opinion, the automated installation is the better alternative. After the end user has chosen what products he wants to be installed, I think it is better that his interaction is not needed.

This makes user's life much easier and as I said before, if the installation is seen as a service to the customer it should not involve much work from his side.

Some guidelines require the possibility for both local and remote installations [Logo Guidelines, 1998] [Logo Handbook, 1998] [Segal & Frieder, 1993]. Remote installations are done, for example in the case of network installations. In this situation, applications should be capable of running directly from the source and computer and user-specific settings should be collected the first time the application is run, not during the installation. The installation should also support administrative additions during network installation [Logo Guidelines, 1998] [Logo Handbook, 1998]. This means that network administrators can add additional components to network installations (such as third-party tools, add-ons, and so forth) [Logo Guidelines, 1998] [Logo Handbook, 1998].

Another situation where there is need for remote installation is with distributed programs. Especially installations of large distributed programs are quite complicated because these applications include hundreds of thousands of individual modules, and the installation should cooperate with other updating systems across administrative domains. [Segal & Frieder, 1993]

There can also be requirements for the delivery media from which the software is installed to the end users machine. Jacsó [1992] prefers products which have installations of the "load-and-go" type or which are stored on the CD-ROM. If no other alternative is available, he also accepts the floppy disks as delivery media [Jacsó, 1992]. Hoek et al. [1997] state more generally that the applications should be available through multiple channels, e.g. via FTP-sites, Web pages, a developer-controlled mailing list etc. thus leaving the choice which alternative to use to the customer. In my opinion, it is not possible to say in general, what is the right form to deliver the software product. The delivery media requirements depend on hardware capabilities and the number of customers, for example. The customer needs should be, however, considered and if there is a need for varying types of delivery media, they should be offered.

It must be noted that the requirements for alternatives concern all types of installations: commissionings, upgrades, and uninstallations. [Schiele, 1995] [Logo Guidelines, 1998] [Logo Handbook, 1998]

4.2.4 Scheduling and downtime

The system administrators are naturally interested in the duration of an installation and if special personnel is needed to accomplish it the timing is also a very important factor. In my opinion, when the duration of the installation is estimated, it is better to overestimate than underestimate. Some time should be reserved for solving special problems, especially in the case of very large systems. Unfortunately, in real life problems do occur and their nature may be totally unpredictable. [Tuominen, 1998]

Even though the installation process should be made as short as possible, it should not be paid with lower quality. To reduce the time spent in the installation, it can be designed so that several things can be done in parallel. Thus, in a problem situation, the installation can be continued even though the problem is not yet solved [Tuominen, 1998].

The usual software maintenance techniques including installations are performed by stopping the whole system, installing the new version, and finally resuming the whole system execution [Amador, *et al.*, 1991]. Under this approach, during a certain period of time (often referred as downtime) the system is not operational [Amador, *et al.*, 1991]. However, there are many systems which do not tolerate any downtime because it causes economic, operational [Segal & Frieder, 1993] and safety problems [Amador, *et al.*, 1991].

If the system does not tolerate any downtime it should be possible to update the system dynamically [Amador, *et al.*, 1991] [Segal & Frieder, 1993] [Fabry, 1976]. This means that the system can replace parts of its code dynamically, while the rest of the system continues working execution [Amador, *et al.*, 1991]. It can be easily noticed that the concept of downtime concerns mainly upgrades and not commissionings or uninstallations. Dynamic updating also requires a lot from the application to be installed. Some software-based solutions for dynamic updating are discussed in Section 4.3.4.

If the downtime cannot be totally avoided, the whole upgrade process should be designed so that the time the system is not operational is minimised. This can be done for example by hardware-based solutions which are typically used in systems that need redundant hardware to provide fault-tolerance (e.g. telecommunications). In this solution an entire running program is dynamically updated with a second system identical to the one executing the older version of the program. The downtime is needed only when the two different running systems are merged back together or when the software shared between the systems (such as a database) is upgraded. In the latter alternative, the downtime is usually longer but the shared data is not duplicated at any stage of the upgrade. This way the data is kept consistent more easily between the two running systems. In general, if this upgrade technique is used, some work in progress may be lost unless the data cannot be buffered temporarily in another place. [Segal & Frieder, 1993]

There are also some ways which make the downtime more tolerable if it cannot be totally avoided. First of all, the customer should be informed about the downtime and its probable duration, which, however, can vary according to the different configurations customers have [Havulinna, 1998]. In addition, the accurate timing of the downtime should be discussed with the customer. Very often the downtime can be postponed without harming the actual upgrade and many customers have certain

periods of time when the system to be upgraded is not in such heavy use, e.g. in the nights and during the weekends.

The upgrade procedure can also be designed so that only some of the operations and services offered by the system are out of use during the upgrade. Thinking about the whole system, this is not actual downtime. However, the customer should be informed which operations are in use in different stages of the upgrade. [Havulinna, 1998]

Table 3. Cost of downtime [**Toigo, 1996**]

Business	**Hourly Financial Impact (US dollars)**
ATM Service Fees	$12,000-$17,000
Online Network, Connect Fees	$23,500-$27,000
Package Shipping, Service Requests	$24,500-$32,000
Cellular (new), Service Activation	$38,000-$44,000
900 Number Services	$54,000-$70,000
Telephone Ticket Sales	$56,000-$82,000
Catalog Sales Centres (Large Retailers)	$60,000-$120,000
Airline Reservation Centres	$67,000-$112,000
Pay-Per-View Services	$67,500-$233,000
Home Shopping Channels	$87,500-$140,000
Infomercial 800, Number Promotions	$175,000-$224,000
Credit Card Sales, Authorisations	$2.2-$3.1 million
Brokerage (Retail)	$5.6-$7.3 million

Source: Contingency Planning Research Inc.'s Annual Disaster Impact Research

Table 3 represents the range of the financial impact, which would result from a one-hour or more outage in different businesses. The financial loss can be only several thousands of US dollars or it can scale to millions of dollars. Obviously, if it is a question of millions of dollars, the customer wants to keep the downtime to a minimum. It should be noted that this table gives only a clue about the financial impact. The safety problems occurring due to downtime can be very severe although their impact can not be measured financially.

4.2.5 Organisational requirements

The company or organisation developing the software product should provide support for installations [Jacsó, 1992]. It must be ensured that both the level and the availability of the support are appropriate [Tuominen, 1998]. In addition to the "human" support, it is also useful to offer a set of tools [Tuominen, 1998], a testing

program or diagnostics messages [Jacsó, 1992] which help problem solving during the installations.

In larger organisations it is vital to clarify responsibilities and ensure fluent information flow between different operational groups dealing with the installations. For example, a feedback mechanism must be provided from people performing installations to people designing installations. Although system administrators usually carry out the installations of smaller software products, for more complicated software systems it might necessary to define who are allowed to accomplish the installations and provide proper training for these people. [Tuominen, 1998]

When special personnel for installations are needed, the personnel usually comes from the software company which developed the software product. This can be seen also as a service for the end user. In my opinion, if the installation is complex and time consuming, the end user companies do not want to allocate any resources for the task. It is easier to let the software vendor do all the work.

4.3 Requirements for applications

Different authors list varying characteristics of applications that make a software product more installable, maintainable or modifiable and thus can be seen as requirements for applications with regard to installations. Lehman [1978] states that the first law of program evolution leads to the requirement of changeability. Pressman [1994] calls this property "effective modularity". Also Oskarsson sees system modifiability as the most important quality but in terms of such system properties as simplicity, observability, extensibility and stability. Sommerville [1982] and Schneidewind [1987] use the term "maintainability". Ghezzi et al. [1991] separate maintainability into two different properties, repairability and evolvability, and also represent the concept "anticipation of change". In addition, Belady and Lehman [1979] stress the importance of total comprehension of a system.

In the following Chapters, some recommended characteristics for an easily installable application are introduced more thoroughly and their effect on installability is discussed. Also some instructions, on how to implement the required characteristics, are presented.

4.3.1 System modifiability

According to Oskarsson modification is an important activity in handling large software systems that are intended for large areas of application and many different environments. This is because flexibility can be achieved either by making the software general (e.g. through parameterisation), or by making it modifiable. Oskarsson claims, and I agree with him, that the latter method is the best way to achieve flexibility. [Oskarsson, 1982]

Both methods are useful, of course, but when using parameterisation the designer should think about all possible modifications which could possible be required and allocate a parameter for them. In my opinion, this is quite an impossible task and so it is easier to make the software generally modifiable although still it is not possible to implement all the changes in the future.

The key to software modifiability is modularization. The ideal situation defined by Oskarsson is that a software system consists of a set of well isolated modules of a suitable size. In addition, it should be possible to implement any individual new requirement by simply changing a module or adding a new module to be connected in a prepared way. The reason for this wish is quite obvious. The human ability to understand how to implement a change depends very much on how easily one can find the information needed. If a change can be implemented in one module, this means that one can find most of the information needed there. In large systems where changes in different parts have to be performed by different persons, this is still more important. A successful modularization is one way to achieve this. [Oskarsson, 1982]

Oskarsson discusses the software system modifiability in terms of modification activities (see Section 2.2.1), modification tasks (see Section 2.3) and system properties. The system properties that support the modification efforts can be combined into four properties: simplicity, observability, extensibility and stability. The difference between system and program modifiability should be noted. "System modifiability" refers to ease of change through addition of modules or exchange of modules, as opposed to "program modifiability", meaning ease of change within modules, in the level of source program statements. [Oskarsson, 1982]

Oskarsson [1982] uses the term *simplicity* for simplicity of structure. In general, this means that the structure is easy to handle because it can be attended to, one part at a time.

Observability is the ability to easily perceive how and why actions occur. Observability is enhanced by simplicity and by such factors as how well implementations of different capabilities are separated, and how well the software structure reflects the structure of the functions and environment of the system. Observability helps to locate the changes, and also supports testing. [Oskarsson, 1982]

Extensibility is the extent to which the program can support extensions of critical functions. Two important factors are the generality of software units, i.e. the reusability of software units, and to what extent the system is prepared for certain types of changes. The system can prepare for changes, for example through parameterisation or prepared interfaces. However, it should be noted that there are probably cases where it is better to plan for redesign than to design all modules to be reusable. [Oskarsson, 1982]

Stability is the resistance to the amplification of changes in the program, i.e. absence of ripple effects. Logical stability is the locality of changes, i.e. good stability

means that changes in the implementation of a certain capability have little effect on other capabilities of the system. [Oskarsson, 1982]

Pressman has an idea very similar to Oskarsson's modularization that he calls *effective modularity*. Pressman claims that effective modularity is a key to good design and thereby the key to software quality. Also his arguments in the favour of modularity have a strong correspondence with Oskarsson's statements although he does not present them in such a structured manner as Oskarsson does. According to Pressman effective modularity (i.e. independent modules) makes the software development easier because functions are separated and interfaces are simplified. Thus, an effective modular design reduces complexity, i.e. increases simplicity, and facilitates change. Independent modules are easier to maintain and test because secondary effects caused by modifications are limited, error propagation is reduced, and reusable modules are possible. These last statements are close to Oskarsson's concepts of "observability, "extensibility and "stability". [Pressman, 1994]

According to Belady and Lehman comprehension of the system as a whole is essential to its effective application and maintenance. This characteristic of understandability is very closely related to Oskarsson's observability. Any interaction with the system, whether for usage or for modification, requires a view of the system as a whole, as an entity. It demands knowledge of the reaction of the system in its entirety, as well as that of each of the parts. The individual implementing a change must know and be aware of the total consequences of his action over the entire system even though the change is done in the code level. This need for simultaneous awareness, at both the global and the lowest levels of detail, is due to the complex and largely invisible structure of system-communication. [Belady and Lehman, 1979]

4.3.2 Maintainability and anticipation of change

Ghezzi et al. include maintainability in the most important qualities of software process and product. Maintainability is seen as two separate qualities, repairability and evolvability although the distinction between these two concepts is not always clear. For example, if the requirement specifications are vague, it may not be clear whether they are fixing a defect or satisfying a new requirement. [Ghezzi *et al.*, 1991]

Software is *repairable* if it allows the fixing of defects with a limited amount of work. Ghezzi et al. also see the importance of modularization. They state that a repairable software product consists of well-designed modules. Thus, it is much easier to analyse and repair than a monolithic one. [Ghezzi *et al.*, 1991]

Software is *evolvable* if it allows changes that enable it to satisfy new requirements. Evolvability is achieved by modularization which is both a product- and process-related quality. In terms of the latter, the process must be able to accommodate new management and organisational techniques, changes in engineering education, etc. As the cost of software production and the complexity of applications grow, the

evolvability of software assumes more and more importance. Indeed, to overcome problems of the continuous changes, the initial design of the product, as well as any succeeding changes, must be done with evolvability in mind. Thus, evolvability is one of the most important software qualities. [Ghezzi *et al.*, 1991]

According to Ghezzi et al., *anticipation of change* is a principle that we can use to achieve evolvability and it is perhaps the one principle that distinguishes software the most from other types of industrial productions. However, the ability of software to evolve does not come for free. It requires special effort to anticipate how and where the changes are likely to occur. When likely changes are identified, special care must be taken to proceed in a way that will make future changes easy to apply. [Ghezzi *et al.*, 1991]

Software engineers must realise the importance of design for change. A common mistake is to design a system for today's requirements, paying little or no attention to likely changes. A consequence of this approach is that a design may turn out to be extremely difficult and costly to adapt to changes, and it will have to be redone almost completely in order to incorporate even seemingly minor changes. Another unfortunate consequence is that in the process of trying to accommodate changes, the designer may have to clutter the initial structure, resulting in an application that is more and more difficult to maintain and that inspires little confidence in its reliability. [Ghezzi *et al.*, 1991]

Ghezzi et al. present six types of most common changes which should be prepared for. The changes may fall under perfective and adaptive maintenance. The types of changes are change of algorithms, change of data representation, change of underlying abstract machine, change of peripheral devices, change of social environment, and incrementality. [Ghezzi et al., 1991]

Ghezzi et al. note that anticipation of change also affects the management of the software process. Depending on the anticipated changes, managers must estimate costs and design the organisational structure that will support the evolution of the software. In addition, anticipation of change requires appropriate tools to be available to manage the various versions and revisions of the software in a controlled manner. The discipline that studies this class of problems is called "configuration management". [Ghezzi *et al.*, 1991]

Also Oskarsson discusses the subject of prepared and unprepared changes. Oskarsson claims that the designer striving to simplify foreseeable changes involve various risks. First, when the change is finally needed, the requirement may be different to what was originally imagined, so the preparation may have been in vain. Secondly, unprepared changes of the system cannot be avoided, and when the prepared change is finally to be effectuated, an unprepared change may very well have invalidated the preparation. Thirdly, the preparation may never be needed. In this case it is an unnecessary burden, making the system unnecessarily complex. On the other

hand Oskarsson points out that the unprepared changes are simplified if the modularization is done so that the modules themselves are easy to understand and change, and there is little coupling between modules. Also, according to Oskarsson a proper regard at system design for the application area can support unprepared changes. [Oskarsson, 1982]

However, Oskarsson discovered that it is difficult to create a system structure where all types of change can be implemented in a simple way. Simplifying one type of change often means complicating other types of change. The task for the system designer is to find a structure where the most common and most expensive types of changes are easily implemented, while not unnecessarily complicating the introduction of other, rarer types of change. [Oskarsson, 1982]

4.3.3 Implementing installability

What means do we have, other than the anticipation of change, to achieve at least some or all the wanted characteristics: simplicity, observability, extensibility, stability, understandability, repairability, and evolvability? As we can see from the arguments above, modularization, i.e. the structure of an application, is one of the most important things to make a software product more maintainable and installable. Other important factors are use of standardised methods (see Section 4.3.3.2), proper documentation (see Section 4.3.3.2) and management activities (see Section 4.3.3.3).

4.3.3.1 Modularization

Lehman [1978] states that the structure is very crucial for changeability, and thus to maintainability, because it makes a program understandable by breaking it, and the processes it controls, into meaningful parts and in a meaningful pattern. The program design should exhibit a hierarchical organisation that makes intelligent use of control among components of software and it should contain distinct and separable representation of data and procedure [Pressman, 1994]. It should be noted that to avoid the difficulties in maintenance the program structure must not only be created but must also be maintained [Lehman, 1978]. The management however chooses whether to follow a strategy of continuous or discrete structural maintenance or a combination of two s[Lehman, 1978]. It is not possible to demonstrate that either of these two strategies is in general better from the economic or technical point of view [Lehman, 1978].

In addition to structure, i.e. the general relationships between substructures, the interfaces between the subsystems must be completely defined and their properties fully specified [Lehman, 1978]. Equally the subsystems, as functional components of the program, must also be fully specified [Lehman, 1978]. Functional components, i.e. modules, must be designed so that each module addresses a specific and logical

subfunction of requirements and has a simple interface when viewed from other parts of the program structure [Pressman, 1994]. In addition, these modules should be as independent as possible from each other [Pressman, 1994] [Schneidewind, 1987] [Sommerville, 1982].

Merely increasing the number of modules, however, does not make a more maintainable product [Ghezzi *et al.*, 1991]. The right module structure with the right module interfaces must be chosen to reduce the need for module interconnections [Ghezzi *et al.*, 1991]. The right modularization, achieved with the help of information hiding and abstraction, promotes maintainability by allowing errors to be confined to few modules, making it easier to locate and remove them [Ghezzi *et al.*, 1991] [Pressman, 1994]. Otherwise it becomes impossible to foresee fully the effect of any local change on other parts of the program and therefore on the detailed behaviour of the program in execution [Lehman, 1978]. In other words, it becomes very difficult to change the program without side effects that modify its behaviour in an unplanned fashion [Lehman, 1978].

According to the principle of *information hiding*, the modules should be specified and designed so that information (procedure and data) contained within a module are inaccessible to other modules that have no need for such information. Hiding implies that modularity can be achieved by defining a set of independent modules that communicate with one another only that information which is necessary to achieve software function. The use of information hiding provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedures are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software. [Pressman, 1994]

Also Belady [1980] stresses the importance of locality in software changes. The more predictable the changes are, the easier it is to locate the consequences of planned modifications [Belady, 1980]. According to Belady particularly *data abstraction* eases locating and localising changes. Also Pressman [1994] states that the abstraction helps to define the procedural (or informational) entities that comprise the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module [Pressman, 1994].

Pressman measures functional independence with two qualitative criteria: cohesion and coupling. *Cohesion* is a measure of the relative functional strength of a module. *Coupling* is a measure of the relative interdependence among modules. An easily maintainable and reusable module has high cohesion and low coupling. [Pressman, 1994]

Cohesion is a natural extension of the information hiding concept. A cohesive module performs a single task within a software procedure, requiring little interaction with other parts of a program. Stated simply, a cohesive module should (ideally) do

just one thing. It is important to strive for high cohesion and recognise low cohesion so that software design can be modified to achieve greater functional independence. [Pressman, 1994]

Coupling is a measure of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data passes across the interface. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect" caused when errors occur at one location and propagate through a system. [Pressman, 1994]

Pressman makes a difference between external and common coupling because the acceptability of their occurrence varies. *External coupling* occurs when modules are tied to an environment external to the software. For example, I/O couples a module to specific devices, formats and communication protocols. External coupling is essential, but should be limited to a small number of modules within a structure. *Common coupling* occurs when a number of modules reference a global data area. Diagnosing problems in structures with considerable common coupling is time-consuming and difficult. However, this does not mean that the use of global data is necessarily "bad". It does mean that a software designer must be aware of potential consequences of common coupling and take special care to guard against them. [Pressman, 1994]

It should be noted that since Oskarsson's dissertation, software engineering has introduced a method which makes modularization much easier than older design techniques: object-oriented design (OOD). The unique nature of object-oriented design lies in its ability to build upon three important software design concepts just described: abstraction, information hiding, and modularity [Pressman, 1994]. OOD provides a mechanism that enables the designer to achieve all three without complexity or compromise [Pressman, 1994]. Thus, nowadays, when object-oriented methods are commonly used, the implementation of modularization should not be such a difficult task.

4.3.3.2 Standardised methods and documentation

Generally, inadvertent carelessness in design, coding, and testing has an obvious negative impact on the maintainability of a software product [Pressman, 1994]. In all the phases of software development a repeatable and standardised method should be used [Sommerville, 1982]. As already noted before, the more time and effort is spent on design validation and program testing, the fewer errors there are in the program and consequently decreased maintenance costs resulting from error correction [Sommerville, 1982].

Programs written in a high level programming language are usually easier to understand, and hence maintain, than programs written in a low-level language. [Ghezzi *et al.*, 1991] [Sommerville, 1982] It should be taken into account that the way

in which a program is written clearly contributes to its understandability [Sommerville, 1982]. Thus, good *programming style* makes the future modifications easier [Sommerville, 1982].

If a program is supported by *proper documentation*, the task of understanding the program can be relatively straightforward. Consequently, program maintenance costs tend to be less for well-documented systems than for systems supplied with poor or incomplete documentation as was already noted in Section 2.2.3. [Sommerville, 1982]

Documentation must be created and updated continuously to record system features, individual design and implementation decisions, the considerations on which they were based, and the details of interfaces between individual systems. More generally it should provide a permanent, accessible, complete and correct record of innumerable, transient yet possibly significant, interhuman communications. In practice, it is of course rare that any of this is done completely and correctly. [Belady and Lehman, 1979]

In the documentation, special notice should be given to the structure, to the program modules and their interconnections. Several types of structure need to be understood and documented: procedural, control, data, and input/output structure. Also data aliases, data flow, control flow and differences between program versions should be easily displayed and understood. [Schneidewind, 1987]

To avoid problems due to component dependencies Hoek et al. [1997] suggest that the developers must carefully and accurately describe their system, especially in terms of its dependencies on other parts of the system. The problem concerning dependencies occurs at all levels of a hierarchically structured system and they can also cross organisational boundaries [Hoek *et al.*, 1997]. If developers were able to easily describe or annotate dependencies during the software design and implementation, these dependency annotations could be used to better understand (and even automate) what is required to build, install, and/or execute a system [Hoek *et al.*, 1997]. The best situation would be the one without any dependencies between the program modules, although in reality, such a situation is quite impossible to achieve.

It should be noted that not all dependencies are alike. Some dependencies are known at the time the programs are compiled (compile or build-time dependencies) while others are resolved at the runtime (runtime or service usage dependencies) [Ruonavaara, 1997] [Hoek *et al.*, 1997]. There can also be data dependencies caused by common data: relational database, flat files or any other type of external shared data [Ruonavaara, 1997]. The categories of system dependencies are illustrated in Figure 5.

Finding and describing the structure and dependencies is quite an effort and therefore they are not easy to accomplish without proper tools. Thus, there is a clear need for tools which could help in this job. [Hoek *et al.*, 1997] [Schneidewind, 1987]

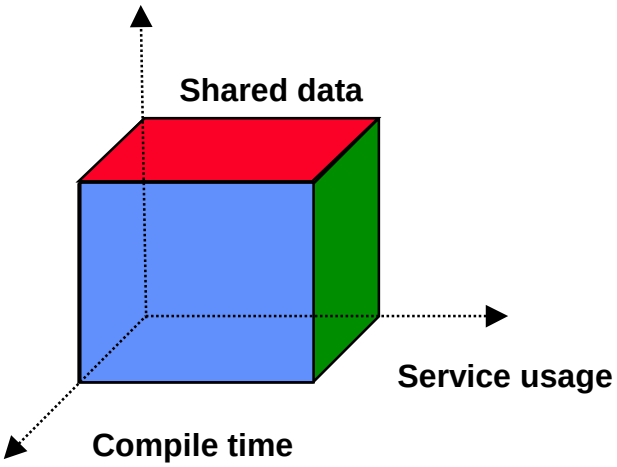

Figure 5. The categories of system dependencies [Ruonavaara, 1997]

4.3.3.3 Management activities

The third and fourth law of software evolution have some implications for the management of software maintenance (see Section 2.1). The interpretation of Lehman's third law as expressing natural properties of the implementation environment as a self-stabilising feedback system, leads to certain rules for successful project management. First, the management should accept as a fact that its project and product has, or will develop in the course of time, certain natural parameters and trends. Secondly, the management should attempt to understand why the trends occur and how they may be overcome. The project and system parameters, such as those described in the published evolution dynamics literature, and estimate project models should be measured modifying the estimates as new information comes to hand. The models should be used to plan further evolutionary maintenance using both long range and cyclic trends. The activity of model identification and interpretation should yield increased insight into, and understanding of, the programming process as practised locally. If this is achieved, it may be used to improve the process gradually, to yield more cost-effective parameters, increased productivity for example. [Lehman, 1978]

As a consequence of Lehman's fourth law of conservation of organisation stability a competent manager planning a spurt of activity to reach some specified target would plan for a subsequent period of lower productivity to allow for corrective action; for the recovery of both system and personnel. The analysis of project history can thus be used as a guiding rule. [Lehman, 1978]

Also the lower level management of maintenance can be improved. This means that maintainers should be involved in design and testing and personnel should be rotated between design and maintenance. The same emphasis should be put on the use of standards in maintenance as design. The use of design tools should be carried over

into maintenance. Configuration management and change request procedures should be used and a liaison between users and maintenance should be established. To motivate the maintenance personnel and thus maximising their productivity the management should strive to rectify the negative image of software maintenance. [Schneidewind, 1987]

Naturally the maintenance is also easier if the person or group that originally developed the software is available. [Pressman, 1994]

4.3.4 Software-based solutions for dynamic updating

A dynamic program-updating system can make it easier to repair bugs or enhance running software without the cost of system shutdown, i.e. with no downtime. Hardware-based solutions were shortly mentioned in Section 4.2.4 (Scheduling and downtime). This section introduces two ideas for software-based solutions. The types of software-based systems include replacement of abstract data types in programs, replacement of servers in client-server systems, updating of distributed programs that use externally specified communication topologies, and updating of programs in procedural languages. [Segal & Frieder, 1993]

The software-based solutions for dynamic updating are very demanding for the underlying software implementation. For example, dynamic replaceability requires that indirection between the program modules that invoke each other is incorporated into a language or its underlying runtime system. The system must also provide techniques for preserving the correctness of a program being updated. The high requirements might be the reason why the research has so far concentrated on creating dynamic updating techniques for specific, well-accepted, and well-understood program structures. [Segal & Frieder, 1993]

Fabry presents two simple requirements which make the dynamic updating possible:

V1. A version number field should be added to each data structure, module etc. The version number should appear in a standard place in the structure. The way in which the version number is referenced should not be changed between different module versions. [Fabry, 1976]

L1. In order to tell when one or more processes are executing the module and using a particular data structure, it will be assumed that the data structure includes locking data and that the modules always lock the data structure before using it. Many processes can access a data structure simultaneously but it must be insisted that a process updating a data structure has exclusive access. [Fabry, 1976]

Amador et al. present a complete design method for software systems including replaceable components. The executable unit (i.e. the process) is selected as the

minimal Software Replaceable Unit (SWRU) that can be dynamically replaced. According to Amador et al. a SWRU may be defined as: a piece of code, data or textual documentation that can be treated and considered as a unit that can be replaced within its operative environment under specific conditions. [Amador *et al.*, 1991]

The requirements for SWRUs are quite strict [Amador *et al.*, 1991]:

S1. SWRU must completely encapsulate their internal state, providing no access to internal state variables.

S2. SWRU must make no reference to any shared components having an internal state. If they share components with others, these shared components must have no internal state.

S3. SWRU must communicate with others through well-defined interfaces, using an explicit way of message passing. This means that no shared variables are allowed.

S4. Each SWRU must have its own flow of control being able to execute on a separate machine of a network. Operations of SWRUs are activated through the control flows.

S5. Each SWRU shall provide specific support for dynamic replacement, including facilities to store and retrieve its internal state.

The internal state of a SWRU is also referred as the checkpoint data which is the set of data that gives the information about the value of the meaningful items at a given time [Amador *et al.*, 1991].

It can be noted that communication is a very important issue for SWRUs and sharing of data, as defined above, is unacceptable. In other words, SWRUs are highly cohesive components, encapsulating functions that are strongly related, and loosely coupled to each other, so that the communication between them may be handled in an efficient way. [Amador *et al.*, 1991]

Each SWRU can accept five replacement operations: Run, Stop, Continue, Store Checkpoint Data and Retrieve Checkpoint Data. The main steps of the replacement process are 1) storing the checkpoint data from the old SWRU, 2) obtaining the checkpoint data by the new SWRU, 3) stopping the old SWRU, 4) starting the new SWRU. The replacement model requires also the existence of a replacement supervisor. For example, the supervisor stores the checkpoint data during the replacement process. [Amador *et al.*, 1991]

Even if we accepted the sharing of data for the SWRUs, the dynamic replacement might still be possible if the SWRUs would use a locking mechanism presented by Fabry (requirement L1). However, the disadvantage of this choice is that the replacement process could last longer because the process should wait for the exclusive access to the shared data component.

4.4 Requirements for installation procedures

In this section the requirements for commissioning and upgrade procedures are concerned. The uninstallation procedures are discussed in Section 4.5.

Mainly four types of requirements were important for commissioning and upgrade procedures. First, the procedures should assume as little as possible about the environment where the software is installed (see Section 4.4.1). Secondly, the location of the software components is not insignificant (see Section 4.4.2). In addition, the replacement of different software components and configurations cannot be done straight forwardly (see Section 4.4.3). Finally, the procedures should be repaired for failures and interruptions from the end user side (see Section 4.4.4).

4.4.1 Better safe than sorry

Before the installation is started, user's hardware and software configuration should be checked [Beiser, 1994] [Logo Guideline, 1998] [Logo Handbook, 1998] to verify that everything needed to successfully install and run [Jacsó, 1992] [Schiele, 1995] the application is available. In general, the software must make no assumptions about the hardware to which it is delivering software [Lawson, 1994].

For example, you should check that there is enough free disk space [Jacsó, 1992] [Logo Guideline, 1998] [Logo Handbook, 1998] [Schiele, 1995] and enough RAM [Jacsó, 1992] and also that the needed third-party software applications are installed on the machine. If there are some hardware or software requirements which are not met, the installation program should give an appropriate warning [Beiser, 1994] [Schiele, 1995]. However, the user must be able to decide whether to continue or interrupt the installation [Schiele, 1995].

All the operations should be checked in advance to ensure that they can be completed successfully. For example, write access to a user's machine should not be automatically assumed. During the installation you should check for the user privilege level and the installation should fail gracefully in the case where a system component cannot be replaced because file system security prevents an existing file from being overwritten. If the user is not an administrator and the application will work but with limited functionality, the installer must warn the user that only limited functionality will be available since they do not have administrator privileges, and the installer must allow them to discontinue the installation. It should be also checked that non-authorised users can not complete the installation. [Logo Guidelines, 1998] [Logo Handbook, 1998].

In general, installation procedures should supply default values [Schiele, 1995], for example installation paths [Logo Guideline, 1998] [Logo Handbook, 1998], but there should be always possibility to change them [Jacsó, 1992] [Tuominen, 1998]. The installation should also be resilient in the face of system customisation [Simmons,

1997]. Installation procedures should query drive locations, file names and directory paths e.g. from the user [Jacsó, 1992] [Schiele, 1995] or in the registry (Windows) [Logo Guidelines, 1998] [Logo Handbook, 1998]. This is because the user might have modified some values even though the installation designer would see them as "defaults" and in general it is more respectful to let the system administrator choose e.g. the destination where the software is installed.

According to Logo Guidelines and Handbook installation procedures should not even assume on which platform you are installing. Of course, this is only if the product can be installed on multiple platforms from a single distribution media. In this situation, the installation procedure must automatically detect the platform. It is not acceptable to ask the user which platform to install on. [Logo Guidelines, 1998] [Logo Handbook, 1998]

4.4.2 Directory structure

It is recommended to create an own directory (structure) for the application to be installed [Schiele, 1995] and copy all the application executables and data files to this same directory on the hard disk [Logo Guideline, 1998] [Logo Handbook, 1998] [Schiele, 1995]. If the platform or operating system has a common place for installing applications, this is just the place where the directory should be created [Logo Guideline, 1998] [Logo Handbook, 1998]. For example, in Windows such a place is the \Program files folder [Logo Guideline, 1998] [Logo Handbook, 1998] and in Unix /opt directory. The executables and files should never be installed in the system directories [Schiele, 1995] or in the root directory [Logo Guideline, 1998] [Logo Handbook, 1998]. All the files installed and directories created during the installation should have descriptive names [Schiele, 1995] and follow the naming rules of the operating system.

The user specific data should be separated from application files and it should be stored to the user's profile folder (Windows), user's home directory (Unix) or equivalent location. All the shared files should be installed in the definite shared locations during network installation. In other words, if a file is shared by multiple users or applications its location should be common and accessible for all the users. [Logo Guideline, 1998] [Logo Handbook, 1998]

Although very often the decision about the location of the software components is done during the installation, it should be noted that the directory structure is not only dependent on installation procedures. Also the software can include some assumptions about the location of its components and the directory structure. Thus, to fulfil this requirement the location of the software components must be thought of at the latest during the implementation phase of the application.

4.4.3 Overwriting and user settings

In the old environment, nothing should be overwritten during the installation, at least not without asking the user first [Beiser, 1994] [Bianco *et al*., 1994] [Jacsó, 1992] [Logo Guidelines, 1998] [Logo Handbook, 1998] [Schiele, 1995] [Simmons, 1997]. This rule applies from user and group ids to file permissions [Simmons, 1997] and overwriting newer versions of a configuration file [Bianco *et al*., 1994] [Jacsó, 1992] [Logo Guidelines, 1998] [Logo Handbook, 1998] [Schiele, 1995].

To avoid problems with overwriting, the configuration files should include a version number and date [Fabry, 1976] [Jacsó, 1992] and installation procedures should determine whether any of the files to be installed are already on the hard disk and replace files only with newer versions [Schiele, 1995].

The installation should not change the standard system environment. In other words, the system files [Jacsó, 1992] [Logo Guidelines, 1998] [Logo Handbook, 1998] should not be changed. Installation procedures should not even copy any files to the system directories during installation [Schiele, 1995]. If the system files must be modified, the changes should be done carefully and conflicts with other applications should be avoided [Jacsó, 1992] [Logo Guidelines, 1998] [Logo Handbook, 1998]. The permission for the modification should be asked [Beiser, 1994] [Jacsó, 1992] or the user should be able to do the changes manually [Jacsó, 1992]. Current files must be backed up with obvious filenames [Beiser, 1994] [Jacsó, 1992] [Simmons, 1997].

User settings should be maintained across application version changes [Logo Guidelines, 1998] [Logo Handbook, 1998]. Examples of user settings include customised toolbars, any dynamic, user-generated lists or other user specific options [Logo Guidelines, 1998] [Logo Handbook, 1998]. It is also recommended that user settings can be moved from machine to machine [Logo Guidelines, 1998] [Logo Handbook, 1998]. If there is need to move user settings or other configuration files from machine to machine, the format of the files should be also thought of. For example, ASCII text files are much more easily moved than more complicated forms of configuration files.

4.4.4 Preparations for failure and cancelling

If something goes wrong during an installation, the system should not be left in a state where neither the new version nor the old (if any) of the software is usable [Bianco, *et al*., 1994]. Either the explanation on every step of the detailed recovery procedure should be provided [Havulinna, 1998] for the system administrator or the recovery should be done automatically [Simmons, 1997]. In the case of failure, the importance of backups [Beiser, 1994] [Jacsó, 1992] [Simmons, 1997] and documentation is very high.

The user should be able to cancel the installation before it is finished [Schiele, 1995] and also in this situation, the old version of the product and the environment before installation should be restored.

4.5 Requirements for uninstallation procedures

In addition to the requirements that the software product should be uninstallable and there should be an uninstaller in the first place [Logo Guidelines, 1998] [Logo Handbook, 1998] [Simmons, 1997], there are also some specific requirements for the uninstallation procedures.

The uninstallation must remove all the files and folders from the hard disk associated to the software product. However, there are certain exceptions for this rule. The user data files and resources that other programs might use should be left in their places. For example, it would be very strange, if the uninstallation of a word processor would remove also all the documents written with the program. Also, if there is any reason to believe that removing a component might harm other applications, it is better to leave it behind. However, the uninstallation must inform about everything left behind. In addition to the application files, the uninstaller must remove also itself during the uninstallation as mentioned before. [Logo Guidelines, 1998] [Logo Handbook, 1998]

If the platform has a registering system for shared resources, it is easier to know which components can be removed. For example, in the Windows environment the registry is this kind of system. During the installation all shared components should be registered, i.e. their count is increased, and correspondingly during the uninstallation the count should be decreased. When the count is zero, it should be safe to remove the component. Of course this system fails if there is even one installation or uninstallation which does not follow the rules. [Logo Guidelines, 1998] [Logo Handbook, 1998]

The uninstallation must also restore all the modifications made to the system, as I pointed out in Section 1.2. This usually means restoring system files. However, the uninstallation procedures should not blindly copy back the modified system files. After reversing the changes made to the system file during the installation of an application, the restored file should be compared to the current system file. If they differ, there have been subsequent changes to the system file that would be lost if the old versions of the system file were simply copied back. [Simmons, 1997]

If the system file restoration is too complex task to be accomplished automatically, it would be wise to inform the user that the tasks could not be done. The user should be informed also about the location of the old system file if he wants to restore it manually. This instructions applies naturally also to other components in the environment which cannot be restored automatically.

The security of the uninstallation must also be taken care of. A user without adequate security permissions must not be able to uninstall an application. In this

situation the uninstallation procedures must provide an appropriate error message. However, a system administrator should be able to uninstall all applications that were installed (by a user or administrator) using the default installation. [Logo Guidelines, 1998] [Logo Handbook, 1998]

4.6 Requirements for installation tools

The references do not direct requirements towards installation tools in particular. However, a few demands can be understood. If there is a common installation tool for the platform, it should be used. For example, Logo Guidelines [1998] and Handbook [1998] recommend that Windows applications should use Windows installer in installations. They also require that an installation tool must have a graphical user interface [Logo Guidelines, 1998] [Logo Handbook, 1998]. More generally, it can be required that an installation tool must be easy to use and it should offer methods appropriate for both the ignorant end user and the sophisticated large site manager [Simmons, 1997].

An important requirement is also that an installation tool should offer a progress indicator [Schiele, 1995]. Although the indicator is definitely a part of the tool this demand indicates that also installation procedures must have a way to communicate their progress to the tool. The same requirement concerns also the message and log files. The functionality of writing to a log or message file must be implemented both into the installation tool and into the procedures.

4.7 Common requirements

Let us also consider the common requirements for applications to be installed, installation procedures, and installation tools. These requirements are often such that they are naturally taken into account in the normal software design but for one reason or another they are forgotten when installation procedures and tools are designed.

Like software applications both installation procedures and tools should provide good documentation [Simmons, 1997] [Tuominen, 1998], be tested properly [Jacsó, 1992] [Logo Guidelines, 1998] [Logo Handbook, 1998] [Tuominen, 1998] and give good and helpful error messages in the case of both internal and external failure [Simmons, 1997] [Schiele, 1995].

The documentation and testing of the applications were already discussed in Section 4.3.3.2 (Standardised methods and documentation). However, the nature of installation documentation is a bit different from it and it is discussed in Section 4.7.1. A short look to the testing is taken in Section 4.7.2 and portability is discussed in Section 4.7.3.

4.7.1 Installation documentation

The documentation of installations is two-fold [Simmons, 1997]. The information about changes is needed both during and after the installation [Bianco *et al.*, 1994] [Logo Guideline, 1998] [Logo Handbook, 1998] [Simmons, 1997].

During the installation process, some dynamic tracking of what is done should be provided. The tracking information should be preserved in a reasonable location which can print messages at install time, logs, message files and sometimes even recordings of the install process. Straight after installation it is good to have such documentation with which the system administrator can determine if the installation is correct. [Simmons, 1997]

Later on, proper documentation can also help the system administrator to keep a track on what software is and is not available on a given machine [Bianco *et al.*, 1994] [Logo Guidelines, 1998] [Logo Handbook, 1998]. Not only does this information prove useful when debugging software incompatibilities, but it is quite handy when the system administrator decides to update all of the software packages to the latest versions available [Bianco *et al.*, 1994].

The installation documentation should cover files added, files changed, and preserved files [Simmons, 1997]. In other words, a clear indication should be provided of what has been installed and where [Bianco *et al.*, 1994] [Simmons, 1997]. From all the files the following information should be available: file ownerships, group memberships, permission modes, and a checksum of all files which should be invariant [Simmons, 1997]. It is very useful also to have a script to determine if any files have changed and how, and if some files have been added or are missing [Simmons, 1997].

Configurations and installations should also be commented straight to the configuration files, if it is possible. For example, in Windows the use of informational keys in the registry is recommended. Particularly it is recommended to document the modifications made to the system files. [Logo Guideline, 1998] [Logo Handbook, 1998]

The quality of the documentation affects also the uninstallation. Simmons [1997] claims that for a product to be uninstallable, there must be careful tracking of the location of all installed files and all modifications made to other system files. Also the explanation for every step of the detailed recovery procedure should be documented if the software installation fails [Havulinna, 1998].

In general, the differences between old and new software versions including hardware configuration, communication protocol, runtime environment, modification to existing software process etc. should be documented as well as every step of the upgrading procedure [Havulinna, 1998]. The former description serves software designers, as well as installation designers and people accomplishing the installations.

4.7.2 Testing

Installation tools and procedures should be tested like any other software product [Havulinna, 1998] [Tuominen, 1998]. Some instructions do take this fact into account. For example, the Logo Guidelines [1998] and Logo Handbook for Windows [1998] provide some testing rules. In addition to this, the installation process itself should include post-checking procedure which tests the result of the installation [Jacsó, 1992]. The testing after installation is very important to verify the correctness of installation [Jacsó, 1992].

4.7.3 Portability

Some requirements concern portability of installation procedures and tools. The portability can be required between different operating systems [Lawson, 1994] [Logo Guidelines, 1998] [Logo Handbook, 1998] or between different hardware platforms [Lawson, 1994].

Logo Guidelines and Handbook offer some useful tips on how to implement the installation if you want the application to be fully functional after an operating system upgrade. You should not place the application executables, configuration files etc. in different locations on different operating systems. You should not install different files on the two operating systems or different versions of files that are common to both operating systems. You should neither make operating system specific calls which are not common in both systems. [Logo Guidelines, 1998] [Logo Handbook, 1998]

Another possibility to make the application and its installation portable to different operating systems is to use the following procedure. The installation should offer an option to the user whether to install all additional binaries that would be required to make the program operate on the other operating system. Thus, the application determines at run time which components to use. [Logo Guidelines, 1998] [Logo Handbook, 1998]

It is also reasonable to note that if the software to be installed is not portable to different environments there is usually no reason why the installation procedures and tool(s) should be.

5. An example environment and related problems

The Nokia NMS/2000 is one of the leading network management systems for managing GSM networks [Aalto, 1995]. It is an example of a large system developed using object technology [Aalto, 1995]. It consists of more than one hundred concurrently running processes with over 2.400.000 LOC (lines of code) written in C++ [Aalto, 1995] [Rajala, 1998]. The development of the system takes place in parallel main projects that each has several sub-projects [Aalto, 1995]. Each main project produces a release, which consists of features that implement an increment to the functionality of the existing system [Aalto, 1995]. The size of an increment is typically 200.000 – 300.000 LOC [Aalto, 1995].

According to the system size categories presented in Section 3.1 Nokia NMS is a very large system (see Table 2, Page 16), although the size of the product is approximately 500 megabytes. However, it should be noted that the sizes of software products have increased during the last 13 years and thus a better criteria for categorisation is the number of programmers, for example. Besides the number of programmers, the Nokia NMS/2000 fulfils also other criteria of a very large system. It is a mission critical system for network operators who use it in building and monitoring their networks [Aalto, 1995]. Therefore, the requirements for software quality are high, and the functionality of the system needs to match closely the business activity of the operators [Aalto, 1995].

The standard Nokia NMS/2000 consists of one to three servers and several operator seats, which can be either application workstations or X terminals. These components are connected to a Local Area Network. The servers consist of a communications server, a database server and a standby server, or combinations of these. [Woods, 1996]

The communications server takes care of the data communication between the network elements and the NMS/2000. The Nokia NMS/2000 uses a relational database to store network management data. The database engine runs on *a database server*. The database is structured to separate tablespaces for different types of data: fault, performance and configuration data each have their own tablespaces. *The standby server* can take the role of either database or communication server in the event of failure of one or the other. This redundancy ensures added security to the system as well as easing the upgrade process. [Woods, 1996]

The application products of the Nokia NMS/2000 are Fault management, Trouble management, Performance management, Radio Network management and System management [NMS Products, 1998]. However, in the software component level this operational division is not visible.

Although the problems have been found to vary from one installation to another [Tuominen, 1998] the main problems of installing the Nokia NMS/2000 are

- the software structure,
- duration of installations (especially in upgrades),
- duration of the downtime,
- long period of time between the releases, i.e. between the installations,
- non-standard customer environments, and
- the lack of competent human resources.

At the moment, the whole system is upgraded at the same time because the software structure does not allow partial installations. However, the slow delivery of the new features to the customer have put high pressure to developing such an installation system where subproducts could be installed separately [Tuominen, 1998]. This would also decrease the duration of the downtime, another serious problem of the current installation system. It should be noted that to make the installations better it is not just enough to develop the installation procedures but also the software must be modified to be more installable.

Exceptions for installations of the whole system are so called change and functional notes which can be delivered to the customer between the releases. The change and functional notes can implement error corrections or totally new features. Although these notes are a useful way to get something delivered fast, they have also made the installations more difficult. Because the change and functional notes do not always register themselves properly during the installation, it is sometimes impossible to find out the software configuration on a server. In such a situation, it is not possible to know what should be installed and if the installation will succeed in the first place. Similar problems occur if the hardware configuration does not follow the expected standards.

It is noteworthy that the financial cost of downtime in the NMS/2000 is not the biggest concern of the customers. However, when the Nokia NMS/2000 is not operational, it is much more difficult for the operator to monitor the network. Thus, if something unexpected happens in the network during the downtime and it is not noticed, the costs of the failure can be very high.

6. A system for dynamic installation of distributed software components

In this chapter, I propose, partly based on discussions with Nummenmaa [1998], a system for dynamic installation of a large software system which consists of distributed software components. The main requirement of the installation system is that the components can be installed one at a time and independently of each other. Thus, also downtime can be minimised. The definition of a component is done from the point of view of installations. I define the component as such a part of application software which is wanted to be separately installable.

An application has to fulfil certain requirements to be installable by using this installation system. I take the ideas of Fabry and requirements for SWRUs (see Section 4.3.4) as the basis of my demands. In addition to the requirements [V1], [L1], and [S1-S5] I require the following things from the application and its components:

V2. The components know their version numbers and the version can be queried from the component. This means that the software configuration can be checked dynamically.

V3. All the dependencies between the components are known (compile time, runtime and common data). Also the external environment is divided into components, i.e. the operating system is a component, and their dependencies are treated equally.

S6. The component is able to inform the system, e.g. the communication system, when its upgrade starts and ends. The system can handle the situation when one of its components is out of use. During the upgrade the messages coming to the upgraded component are buffered and the components sending the messages are informed about the possible delay.

In general, the way the requirements are implemented is not significant regarding the installation system. However, it can be noted that the requirements [L1] and [S5] can be fulfilled by proper use of a database system, for example. It should also be noted that the upgrade of the database system might require downtime even though otherwise it can be avoided.

The installation system is presented in Figure 6. The requirements which make a step possible in the procedure are listed after the step description. If there are no dependencies between the components to be upgraded, only one component is shutdown and upgraded at a time (Steps 3-5). In such a situation, it is also possible to upgrade the components in parallel because the order of the upgrades is insignificant.

Input:

a) An installation I of an application A, which fulfils the conditions [V1-V3], [L1], and [S1-S6]
b) All versions of all components of A
c) The new version numbers of the components required to be upgraded in I

Output:

The procedure will upgrade the required components and other such components which must be upgraded as a consequence of the required upgrades.

Procedure:

1. Find out the current software configuration. [V2]
2. Using version compatibility information compute the components which must be upgraded as a consequence of upgrading the required components. [V1-V3]
3. Shutdown the components which are to be changed. The components inform the communication system that they are not available because of the installation. [L1] [S1-S6]
4. Upgrade the components.
5. Start-up the components. The components inform the communication system that they are available again. [L1] [S5] [S6]

Figure 6. The installation procedure

There are a few things that are worth noticing in the installation system. Because the software configuration can be checked dynamically, different kinds of installations can be calculated when the installation is planned. This is useful if a definite component is wanted to be upgraded but everything else is required to be left as it is. An opposite example is that the operating system version is wanted to be preserved but the other components are wanted to be upgraded to as new versions as possible regarding to the version of the operating system.

The listed requirements for the application can be demanding for an existing software product, although they are quite easily implemented when a new application is built. However, the installation system can still give some benefits if the software can be divided into components even coarsely. This is because the system does not require a certain granularity in the division.

If the installation of the components were standardised, it might be even possible to calculate the estimated installation time, i.e. the time when services provided by the component are not available. In Step 2 when computing the changes which have to be

done, the installation tool could also calculate the time which is needed for the installation.

7. Conclusions

In this thesis, I have studied what guidelines current literature gives for installation designers and what problems there are in installing large systems. I have also proposed a system for dynamic installation of a large software system.

There are some special problems in the installations of large systems caused by the following reasons: large systems have complex interactions to other programs, they are usually mission critical systems and they are developed by a group of persons instead of a single designer. Therefore the installation design requires a group of people to be involved. Also the communication and up-to-date documentation are essential when dealing with large systems.

The requirements for installations were divided into four categories: requirements for installation process, applications to be installed, installation procedures and installation tools.

The main requirements for the installation process itself were simplicity and minimal disruption both for the environment and to the software product. Also alternative options should be provided so that the installation could scale to different situations. The scheduling of the installation should be thought out carefully. It should be noted that not all the software systems tolerate downtime although the usual technique is to stop the whole system for the time of installation. Installations can also have organisational requirements, e.g. proper support and training should be organised.

The applications to be installed should be easily modifiable and maintainable. The ways to achieve these characteristics are anticipation of change, efficient modularization and management activities which support the maintenance work. Although the importance of the standardised methods and documentation are stressed in the application design and implementation, these should not be forgotten in any phase of the installation design and implementation.

The requirements for commissioning and upgrade procedures and for uninstallation procedures were discussed separately. Commissioning and upgrade procedures should not assume anything but instead check before installing what is needed to be done and if it is possible to complete the installation successfully. The directory structure and naming of software components should be done according to the rules of the environment. The procedures should be extremely careful about which software components can be overwritten and that user modifications are maintained across version changes. The commissioning and upgrade procedures should also be prepared for user interruptions and for failure in the installation. Important for the uninstallation procedures is that they remove all the software components and restore all the modifications made to the environment during the installation. However, there are some exceptions to this rule which should be considered.

Even though there were not so many special requirements for installation tools, one of them came up very clearly: the tool should be easy to use. It should also support some operations whose implementation must partly be made also to the installation procedures. These operations are progress indicator and functionality to write a log file.

Although the requirements for the installation process, procedures and tools are very important, it should be noted that the basis for a successful installation is the application to be installed. Schneidewind [1987] said of maintenance work that the main problem in carrying out maintenance is that we cannot maintain a system, which was not designed for maintenance. In the same way it could be said of installations that if the software product is not made installable it is not possible to install it easily. Thus, the requirements for the installation process should be thought of already when the system definition and application design starts. This is particularly important if the application is wanted to be upgraded dynamically. The installation can rarely fix the faults done during the software design and implementation.

References

[Aalto, 1995] Juha-Markus Aalto, Challenges in applying objects to large systems. In: Juhani Iivari, Kalle Lyytinen and Matti Rossi (eds.), *Advanced Information Systems Engineering: 7th International Conference, CAiSE '95, Lecture Notes in Artificial Intelligence* **932** (1995), Springer, 154 -167.

[Amador *et al.*, 1991] Jorge Amador, Belén de Vicente and Alejandro Alonso, Dynamically replaceable software: A Design Method. In: Axel van Lamsweerde and A. Fugetta (eds.), *Proc. of 3rd European Software Engineering Conference*, 211-228.

[Beiser, 1994] Karl Beiser, Moving on up: CD-ROM upgrade paths and problems, *Online* **18**, 5 (Sep./Oct. 1994), 116-118.

[Belady, 1980] L.A. Belady, Modifiability of large software systems. In: M.M. Lehman and L.A. Belady (eds.), *Program Evolution*. Academic Press, New York, 1985, 355-373.

[Belady and Lehman, 1979] L.A. Belady and M.M. Lehman, The characteristics of large systems. In: Peter Wegner (ed.), *Research Directions in Software Technology*, MIT Press, Cambridge, 1979, 106-138.

[Bianco *et al.*, 1994] David J. Bianco, Travis L. Priest and David E. Cordner, Cicero: A package installation system for an integrated computing environment. Draft, 1994. Available as *http://ice-www.larc.nasa.gov/ICE/doc/Cicero/cicero.html*.

[CCI Dictionary, 1998] Computer Currents Publishing Corporation, Computer Currents On-line Dictionary. Available as *http://www.currents.net/resources/dictionary/index.html*.

[Curtis *et al.*, 1988] Bill Curtis, Herb Krasner and Neil Iscoe, A field study of the software design process for large systems. In: *Communications of the ACM*, **31**, 11 (November 1988), 1268-1287.

[Fabry, 1976] R.S. Fabry, How to design a system in which modules can be changed on the fly? In: *Proc. of 2nd International Conference on Software Engineering*, 1976, 470-476.

[Fairley, 1985] Richard Fairley, *Software Engineering Concepts*. McGraw-Hill, Singapore, 1985.

[Ghezzi *et al.*, 1991] Carlo Ghezzi, Mehdi Jazayeri and Dino Mandrioli, *Fundamentals of Software Engineering*. Prentice-Hall, Englewood Cliffs, 1991.

[Jacsó, 1992] Péter Jacsó, *CD-ROM Software, Dataware, and Hardware: Evaluation, Selection, and Installation*, Libraries unlimited, Englewood, 1992.

[Havulinna, 1998] Private discussion with Juha Havulinna, April 1998.

[Hoek *et al*., 1997] André van der Hoek, Richard S. Hall, Dennis Heimbigner and Alexander L. Wolf, Software release management. In: Mehdi Jazayeri and Helmut Schauer (eds.), *Software Engineering: Proc. of 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lecture Notes in Computer Science* **1301** (1997), Springer, 159-175.

[Lawrence, 1982] M.J. Lawrence, An examination of evolution dynamics. In: *Proc. of 6th International Conference on Software Engineering*, 1982, 188-196.

[Lawson, 1994] John R. Lawson, Automatic, network-directed operating system software upgrades: a platform-independent approach. In: *Digital Technical Journal*, **6,** 4, (Fall 1994), 89-100.

[Lehman, 1978] M.M. Lehman, Laws of Program Evolution – Rules and Tools for Programming Management. In: M.M. Lehman and L.A. Belady (eds.), *Program Evolution*, Academic Press, New York, 1985, 247-274.

[Lehman, 1978b] M.M. Lehman, On Understanding Laws, Evolution and Conservation in the Large –Program Life Cycle. In: M.M. Lehman and L.A. Belady (eds.), *Program Evolution*, Academic Press, New York, 1985, 375-392.

[Lehman, 1982] M.M. Lehman, Program evolution. In: M.M. Lehman and L.A. Belady (eds.), *Program Evolution*, Academic Press, New York, 1985, 9-37.

[Lientz and Swanson, 1980] Bennet P. Lientz and E. Burton Swanson, *Software Maintenance Management*. Addison-Wesley, Reading, 1980.

[Logo Handbook, 1998] Microsoft Corporation, Designed for Microsoft Windows NT and Windows 95: Logo handbook for software applications. Version 3.0b, March 3, 1998. Available as *http://www.microsoft.com/windows/downloads/bin/winlogo/logo_handbook.exe*.

[Logo Guidelines, 1998] Microsoft Corporation, Designed for Microsoft Windows: Logo guidelines for Windows NT 5.0 and Windows 98 applications. Draft, June, 1998. Available as *http://www.microsoft.com/windows/downloads/bin/winlogo/logo.exe*.

[NMS Products, 1998] Nokia NMS/2000 Application Products, NMS Marketing Material. Nokia Telecommunications, Network Management Systems, March 1998. Available as *http://wwwalltr.ntc.nokia.com/nms/marketing/files/material/2000_Product_2.ppt*.

[Nummenmaa, 1998] Private discussion with Jyrki Nummenmaa, November 1998.

[Oskarsson, 1982] Östen Oskarsson, *Mechanism of Modifiability in Large Software Systems*. Linköping Studies in Science and Technology, Dissertations, **77**, Linköping 1982.

[Pressman, 1994] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*. European edition adapted by Darrell Ince, McGraw-Hill, London, 1994.

[Rajala, 1998] Pasi Rajala, *T10 Main Project Final Report*, Version 1.0. Nokia Telecommunications, Network Management Systems, Internal document, November 1998.

[Ruonavaara, 1997] Markku Ruonavaara, Adding architecture to design. Position paper at ECOOP'97, FAMOOS Workshop on Object-Oriented Software Evolution and Re-engineering. June 1997. Available as *http://www.fzi.de/ecoop97ws8/ruonavaara.ps*.

[Segal and Frieder, 1993] Mark E. Segal and Ophir Frieder, On-the-fly program modification: systems for dynamic updating. *IEEE Software* **10**, 2 (March 1993), 53-65.

[Schneidewind, 1987] Norman F. Schneidewind, The state of software maintenance. *IEEE Transaction on Software Engineering* **SE-13**, 3 (March 1987), 303-310.

[Schiele, 1995] Teri Schiele, Windows 95 application setup guidelines for independent software vendors. May, 1995. Available as *http://www.microsoft.com/win32dev/guidelns/setup.htm*.

[Sommerville, 1982] Ian Sommerville, *Software Engineering*. Addison-Wesley, London, 1982.

[Setälä, 1998] Arto Setälä, *Nokia NMS Core Products: Description of P2.2 Commissioning and Upgrade Procedures*, NTC TAN 1615/1. Nokia Telecommunications, Internal document, May 1998.

[Simmons, 1997] Steve Simmons, Software design for installability. April 18, 1997. Available as *http://lokkur.dexter.mi.us/~scs/techwriting/install.html*.

[Swanson, 1976] E. Burton Swanson, The dimensions of maintenance. In: *Proc. of 2nd International Conference on Software Engineering*, 492-497.

[Toigo, 1996] Jon William Toigo, Don't crap out on disaster recovery planning. December 1, 1996. Available as *http://www.hppro.com/12dec/toigo1012.htm*.

[Tuominen, 1998] Juha Tuominen, *Upgrade Task Force: Final Report*. Nokia Telecommunications, Network Management Systems, Internal document, January 1998.

[Woods, 1996] Mark Woods, *Nokia NMS/2000 for Managing Cellular Networks*, NTC TAN 0449/5en. Nokia Telecommunications, September 1996.